Embedded Systems
Building Blocks,
Second Edition

Complete and Ready-to-Use
Modules in C

Jean J. Labrosse

R&D Books
Lawrence, KS 66045
To my loving and caring wife and best friend, Manon, and to our two lovely children, James and Sabrina.
# Table of Contents

**Preface** ............................................................................................................ xiii
  - Goals ........................................................................................................... xiv
  - Intended Audience ..................................................................................... xiv
  - Portability .................................................................................................... xiv
  - What Will You Need to Use this Book? ....................................................... xiv
  - Acknowledgments ....................................................................................... xv

**Introduction** ...................................................................................................... xvii
  - Figure, Listing, and Table Conventions ........................................................ xvii
  - Source Code Conventions ......................................................................... xviii
  - Chapter Contents ......................................................................................... xix
  - Web Site ....................................................................................................... xx
  - Bibliography ................................................................................................. xxi

**Chapter 1**  
**Sample Code** ............................................................................................... 1

- 1.00 Installing Embedded Systems Building Blocks ........................................ 1
- 1.01 How Each Chapter Is Organized ............................................................. 2
- 1.02 ENCLOSES_H ......................................................................................... 3
- 1.03 Complex Independent Data Types .......................................................... 3
- 1.04 CFG.C and CFG.H .................................................................................. 4
- 1.05 Global Variables ..................................................................................... 4
- 1.06 OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() ........................................ 6
- 1.07 ESB9 Sample Code .................................................................................. 6
- 1.08 Bibliography ............................................................................................ 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>Footnote/Background Systems</td>
<td>62</td>
</tr>
<tr>
<td>2.01</td>
<td>Critical Section of Code</td>
<td>63</td>
</tr>
<tr>
<td>2.02</td>
<td>Resource</td>
<td>63</td>
</tr>
<tr>
<td>2.03</td>
<td>Shared Resource</td>
<td>63</td>
</tr>
<tr>
<td>2.04</td>
<td>Multitasking</td>
<td>63</td>
</tr>
<tr>
<td>2.05</td>
<td>Task</td>
<td>63</td>
</tr>
<tr>
<td>2.06</td>
<td>Context Switch (or Task Switch)</td>
<td>65</td>
</tr>
<tr>
<td>2.07</td>
<td>Kernel</td>
<td>65</td>
</tr>
<tr>
<td>2.08</td>
<td>Scheduler</td>
<td>66</td>
</tr>
<tr>
<td>2.09</td>
<td>Non-Preemptive Kernel</td>
<td>66</td>
</tr>
<tr>
<td>2.10</td>
<td>Preemptive Kernel</td>
<td>67</td>
</tr>
<tr>
<td>2.11</td>
<td>Reentrancy</td>
<td>68</td>
</tr>
<tr>
<td>2.12</td>
<td>Round-Robin Scheduling</td>
<td>70</td>
</tr>
<tr>
<td>2.13</td>
<td>Task Priority</td>
<td>70</td>
</tr>
<tr>
<td>2.14</td>
<td>Static Priorities</td>
<td>70</td>
</tr>
<tr>
<td>2.15</td>
<td>Dynamic Priorities</td>
<td>71</td>
</tr>
<tr>
<td>2.16</td>
<td>Priority Inversions</td>
<td>71</td>
</tr>
<tr>
<td>2.17</td>
<td>Assigning Task Priorities</td>
<td>73</td>
</tr>
<tr>
<td>2.18</td>
<td>Mutual Exclusion</td>
<td>75</td>
</tr>
<tr>
<td>2.19</td>
<td>Deadlock (or Deadly Embrace)</td>
<td>82</td>
</tr>
<tr>
<td>2.20</td>
<td>Synchronization</td>
<td>82</td>
</tr>
<tr>
<td>2.21</td>
<td>Event Flags</td>
<td>84</td>
</tr>
<tr>
<td>2.22</td>
<td>Intertask Communication</td>
<td>85</td>
</tr>
<tr>
<td>2.23</td>
<td>Message Mailboxes</td>
<td>86</td>
</tr>
<tr>
<td>2.24</td>
<td>Message Queues</td>
<td>87</td>
</tr>
<tr>
<td>2.25</td>
<td>Interrupts</td>
<td>88</td>
</tr>
<tr>
<td>2.26</td>
<td>Interrupt Latency</td>
<td>88</td>
</tr>
<tr>
<td>2.27</td>
<td>Interrupt Response</td>
<td>89</td>
</tr>
<tr>
<td>2.28</td>
<td>Interrupt Recovery</td>
<td>90</td>
</tr>
<tr>
<td>2.29</td>
<td>Interrupt Latency, Response, and Recovery</td>
<td>90</td>
</tr>
<tr>
<td>2.30</td>
<td>ISR Processing Time</td>
<td>91</td>
</tr>
<tr>
<td>2.31</td>
<td>Nonmaskable Interrupts (NMIs)</td>
<td>94</td>
</tr>
<tr>
<td>2.32</td>
<td>Clock Tick</td>
<td>94</td>
</tr>
<tr>
<td>2.33</td>
<td>Memory Requirements</td>
<td>96</td>
</tr>
<tr>
<td>2.34</td>
<td>Advantages and Disadvantages of Real-Time Kernels</td>
<td>97</td>
</tr>
<tr>
<td>2.35</td>
<td>Real-Time Systems Summary</td>
<td>98</td>
</tr>
<tr>
<td>2.36</td>
<td>Bibliography</td>
<td>99</td>
</tr>
</tbody>
</table>
### Chapter 3

**Keyboards** ........................................ 101

3.00 Keyboard Basics ......................................... 101
3.01 Matrix Keyboard Scanning Algorithm ................ 103
3.02 Matrix Keyboard Module ................................ 105
3.03 Internals.................................................. 106
3.04 Interface Functions ...................................... 109
    KeyFlush() ........................................... 110
    KeyGetKey() .......................................... 111
    KeyGetKeyDownTime() .................................. 112
    KeyHit() ............................................. 113
    KeyInit() ............................................ 114
3.05 Configuration ........................................... 114
3.06 How to Use the Matrix Keyboard Module ............... 115
3.07 Bibliography ........................................... 119

### Chapter 4

**Multiplexed LED Displays** .............................. 133

4.00 LED Displays ............................................ 133
4.01 Multiplexed LED Display Module ...................... 136
4.02 Internals................................................ 137
4.03 Interface Functions .................................... 140
    DispClrScr() ......................................... 141
    DispInit() .......................................... 142
    DispStatClr() ........................................ 143
    DispStatSet() ........................................ 144
    DispStr() ........................................... 145
4.04 Configuration ........................................... 146
4.05 How to Use the Multiplexed LED Display Module .... 146
4.06 Bibliography ........................................... 148

### Chapter 5

**Character LCD Modules** ................................ 161

5.00 Liquid Crystal Displays ................................ 161
5.01 Character LCD Module .................................. 163
5.02 Character LCD Module, Internals ....................... 165
5.03 Interface Functions .................................... 167
    DispChar() .......................................... 168
    DispClTime() ........................................ 169
    DispClsScr() ........................................ 170
    DispGetChar() ....................................... 171
    DispHorBar() ........................................ 173
    DispHorBarInit() ..................................... 175
### Chapter 6

**Time-of-Day Clock**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DispInit()</td>
<td>176</td>
</tr>
<tr>
<td>DispStr()</td>
<td>177</td>
</tr>
<tr>
<td>5.04 LCD Module Display, Configuration</td>
<td>178</td>
</tr>
<tr>
<td>5.05 LCD Module Manufacturers</td>
<td>178</td>
</tr>
</tbody>
</table>

#### 6.00 Clocks/Calendars

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClkFormatDate()</td>
<td>196</td>
</tr>
<tr>
<td>ClkFormatTime()</td>
<td>198</td>
</tr>
<tr>
<td>ClkGetTS()</td>
<td>200</td>
</tr>
<tr>
<td>CktInit()</td>
<td>201</td>
</tr>
<tr>
<td>CktMakeTS()</td>
<td>202</td>
</tr>
<tr>
<td>CktSetDate()</td>
<td>203</td>
</tr>
<tr>
<td>CktSetDateAndTime()</td>
<td>204</td>
</tr>
<tr>
<td>CktSetTime()</td>
<td>205</td>
</tr>
</tbody>
</table>

#### 6.04 Clock/Calendar Module, Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.05 Bibliography</td>
<td>206</td>
</tr>
</tbody>
</table>

### Chapter 7

**Timer Manager**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TmrClgFunct()</td>
<td>234</td>
</tr>
<tr>
<td>TmrClg()</td>
<td>236</td>
</tr>
<tr>
<td>TmrFormat()</td>
<td>237</td>
</tr>
<tr>
<td>TmrInit()</td>
<td>238</td>
</tr>
<tr>
<td>TmrReset()</td>
<td>239</td>
</tr>
<tr>
<td>TmrSetMST()</td>
<td>240</td>
</tr>
<tr>
<td>TmrSetT()</td>
<td>241</td>
</tr>
<tr>
<td>TmrStart()</td>
<td>242</td>
</tr>
<tr>
<td>TmrStop()</td>
<td>243</td>
</tr>
</tbody>
</table>

#### 7.03 Timer Manager Module, Configuration

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.04 Bibliography</td>
<td>244</td>
</tr>
</tbody>
</table>
Chapter 11

Asynchronous Serial Communications

11.01 RS-232C

11.02 RS-485

11.03 Sending and Receiving Data

11.04 Serial Ports on a PC

11.05 Level-Level PC Serial I/O Module (COM00_PPC)

11.06 Buffered Serial I/O Module (COMMAND)
Table of Contents — xi

11.07 Buffered Serial I/O Module (COMBIOrio) ........................................... 442
  CommGetChar() .................................................................................. 445
  CommInit() ......................................................................................... 447
  CommIsEmpty() ................................................................................ 448
  CommIsFull() .................................................................................... 449
  CommPutChar() .................................................................................. 450

11.08 Configuration ................................................................................. 452

11.09 How to use the COMPC and the COMHPKG Module ......................... 452

11.10 How to use the COMPC and the COMHPDS Module ............................ 453

11.11 Bibliography .................................................................................. 455

Chapter 12  PC Services............................................................................ 495

12.00 Character Based Display ................................................................. 495

12.01 Saving and Restoring DOS’s Context ............................................. 498

12.02 Elapsed Time Measurement ............................................................ 500

12.03 Miscellaneous ............................................................................... 500

12.04 Interface Functions ....................................................................... 501
  PC_DispChar() .................................................................................... 502
  PC_DispClrCol() ............................................................................... 503
  PC_DispClrRow() .............................................................................. 504
  PC_DispStr() .................................................................................... 505
  PC_DispStr() .................................................................................... 506
  PC_DOSReturn() ............................................................................... 508
  PC_DOSSaveReturn() ......................................................................... 509
  PC_ElapsedStart() ............................................................................. 510
  PC_ElapsedStop() ............................................................................. 511
  PC_GetDate() .................................................................................. 513
  PC_GetDate() .................................................................................. 514
  PC_GetKey() .................................................................................... 515
  PC_GetTickRate() ............................................................................ 516
  PC_VectSet() .................................................................................... 517
  PC_VectSet() .................................................................................... 518

12.05 Bibliography .................................................................................. 519

Appendix A  μC/OS-II, The Real-Time Kernel ............................................ 535
  OSInit() .......................................................................................... 537
  OSSemCreate() ................................................................................. 538
  OSSemPend() ................................................................................... 539
  OSSemPost() .................................................................................... 541
Preface

This is the second edition of Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C. This is a book of software modules that you can use to design embedded systems. The modules are some of the most common building blocks of embedded systems: keyboard scanners, display interfaces, timers, and I/Os. Most of the code is written in highly portable C.

Managers will like this book because it can reduce the amount of time, and thus money, required for some of the more repetitive aspects of embedded systems design. Each chapter is independent of the others, allowing you to use only the module(s) you need. Each chapter describes what the module does, how it works and, what services it provides. This information will help you estimate the resources you'll need to implement your product.

What's new in the Second Edition?

I made a number of changes from the first edition. The most notable one is, of course, the hard cover which makes the book more durable. The second major change is that all of the code and examples have been revised to use μC/OS-II. μC/OS-II is a Real-Time Operating System that I wrote and is fully described in my other book, MicroC/OS-II: The Real-Time Kernel, ISBN 0-87930-543-6, R&D Books.

A scaled-down version of μC/OS-II is provided in object form to allow you to run and change the sample code.

I decided to use the Borland C++ compiler V4.51 instead of V3.1 because some of you had indicated that the version 3 tools are no longer available. I also included a makefile to build the sample code instead of relying on the IDE (Integrated Development Environment). The makefile can easily be changed so the code can be compiled for just about any other target processor.

Chapter 1, "Sample Code", has been completely revised. Chapter 2, "Real-Time Systems Concepts", now contains over 10 new pages. For all the building blocks, I now have a section that presents the APIs (Application Programming Interfaces) in a standard format. This allows you to better use the interface functions of each building block.

In the first edition, Appendix F contained all the data sheets of electronic components I used. I decided to move the data sheets on the companion CD-ROM in PDF form to reduce the book size by about 100 pages and save a few years in the process.

In the first edition, I included the execution times of each of the building block interface functions provided in the book. This proved to be tedious and so I decided to drop this in the second edition. Also, the 80386-computer I had used to come up with the execution times was released a few years ago.
Goals

This book is designed to aid embedded systems programmers by providing ready-to-use modules. If the code in this book doesn't match your exact requirements, you can use the code as a starting point. In other words, it is a lot easier to modify code than to start from scratch. The main objective of this book is to save you time.

Intended Audience

This book is for embedded system programmers, consultants, and students interested in embedded systems. I assume you know C and have a minimal knowledge of assembly language. You should also understand microprocessors and have a basic electronics background. The hardware presented in this book is, however, fairly easy to understand. Because the code is written in C, you can apply the concepts presented in this book to a much broader range of microprocessors (assembly language would not be portable).

If you are a student interested in embedded systems, this book will take some of the mysteries out of the unique requirements of embedded system software design by providing you with concrete programming examples. This book will also allow students to build much more complex embedded systems than would otherwise be possible in the classroom.

Portability

The code presented in this book is written in ANSI C and is highly portable. C has been the language of choice for embedded system designs because it has the following features:

• The code is easier to write and understand than code in assembly language.
• The code generated by some C compilers approaches assembly language in efficiency.
• Once written, C code often can be used on different processors. This is not the case for assembly language code.

In many cases, less than 10% of the code uses more than 90% of the CPU time. You can always optimize this time-critical code by using assembly language. The run-time critical code (90% of the code), can still be written in C. If you are still using assembly language to design embedded systems, you should consider obtaining a C compiler and writing portions of your code in C.

Hardware interface functions have been carefully isolated to minimize the amount of work required to adapt the module to your own hardware environment. I have kept the assembly language to a minimum, and in the places where I have used assembly language, I have kept the code as clear and simple as possible.

What Will You Need to Use this Book?

The code supplied with this book assumes you will be using a PC (98/586 minimum) computer running under either Windows 95/98/NT or DOS v4.0 and higher. The code was compiled with Borland International's new called Inprise) C++ v4.51 (see www.borland.com). You should have about 5 Mbytes of free disk space on your hard drive.
Acknowledgments

First and foremost, I would like to thank my wife for her encouragement, understanding, and patience. This book would never have been possible without her. I would also like to thank my children James (age 9) and Sabrina (age 6) for putting up with having just a mom for a few months while I was 'hiding' in my office working on this new edition. I hope one day they will understand. Special thanks to Dr. Bernard Williams and all the fine people at R&D Books for their help in making this book a reality. Finally, I would like to thank you for buying this book and I hope it will live up to your expectations.
Introduction

I’ve been designing embedded systems for more than 17 years. During that time, I’ve noticed that some of the pieces always seem to keep coming back. I have concluded that 50% percent of the code for an embedded product seems to be similar to the previous product. I always seem to need to read analog and discrete signals, output control signals on analog and discrete outputs, provide some form of user interface and thus, I need to read/can keys on a keyboard and put information on a display device of some sort (7-segment numeric display to an LCD module). Most embedded controllers seem to have an asynchronous serial port (i.e., UART, Universal Asynchronous Receiver Transmitter) and interfacing to a laptop seems like a natural thing to do. I also find myself needing to trigger events when a certain amount of time expires, and to keep track of the date and time. Although it was fun and challenging to develop some of these modules at one point in my career, having to do the same thing over again for each new project has become mundane and even unpleasant. I find that the real challenge is to develop application code that makes my products unique. Over the years, I’ve written fairly generic modules to accomplish some of the functions mentioned above. As I use these modules, I optimized and enhanced them, giving me a good collection of Embedded Systems Building Blocks.

As Steve McConnell mentions in his book, Code Complete, “The single biggest way to improve both the quality of your code and your productivity is to reuse good code.” In his fine book, The Art of Programming Embedded Systems, Jack Gansle states that, “It’s ludicrous that we software people reinvent the wheel with every project. ... Wise programmers make an ongoing effort to build an arsenal of tools for current and future projects ... Collect algorithms!”

If you already write software for embedded systems, this book will provide you with portable, ready-to-use code so that you can save time with your next embedded system design. Time to market is becoming just as important (and in some instances, more important) than the cost of the product itself. Reduced time-to-market provides a competitive advantage.

If I can save you days or even weeks of programming time on one of your projects, I will have met my objectives. You might decide to use the code provided in this book for rapid prototyping or as a permanent addition to your final product. All of the modules presented in this book most likely have both...

xvii
ing to do with what makes your product unique. In other words, your application code is what makes your product different. For example, you may need a keyboard scanning routine and an LCD display module in a FAX machine. What you provide in this product is your FAX machine expertise and you shouldn’t have to spend time with keyboard scanning and LCD display details.

It is very difficult to write 100% reusable code. This is especially true for embedded systems because most embedded systems have very unique requirements and most likely limited memory to hold both the executable portion of your code and its data. The code presented in this book is not intended for embedded systems that will be sold in very large volume. This is because large volume applications are very cost sensitive which means you must typically account for just about every single byte of memory (ROM and RAM); my focus was not to save every single byte.

Figure, Listing, and Table Conventions

You will notice that when I reference a specific element in a figure, I use the letter 'F' followed by the figure number. A number in parentheses following the figure number represents a specific element in the figure that I am trying to bring your attention to. ‘F1(2,3)’ thus means look at the third item in Figure 1.2.

Listings and tables work exactly the same way except that a listing starts with the letter 'L' and a table starts with the letter 'T'.

Source Code Conventions

All of the building block objects (functions, variables, #define constants and macros) start with a prefix indicating that they are related to the specific building block. For example, all clock module functions and variables start with C. Similarly, all timer manager functions and variables start with Tvar.

Functions are found in alphabetical order in all the source code files. This allows you to quickly locate any function.

You will find the coding style I use is very consistent. I have been adopting the L&R style for many years. However, I did add some of my own enhancements to make the code (I believe) easier to read and maintain. Indentation is always 4 spaces, tabs are never used, always at least one space around an operator, comments are always to the right of code, comment blocks are used to describe functions, etc.

I also use and combine acronyms, abbreviations, and mnemonics (AAMs) to make function, variable, and #define names in a hierarchical way (see Appendix C).
Figure 1.1: A block diagram representing the key areas covered in this book.

Figure 1.1 is a block diagram representing the key areas covered by this book. Even though the building blocks shown in the figure interact mostly with hardware, I have carefully isolated hardware-dependent code to a few easy-to-change functions or constants. This makes the code easy to port to your own environment. Also, I avoided using assembly language except when absolutely necessary.

Chapter Contents

Each chapter describes one or more of the building blocks shown in the figure. The building blocks are mostly independent of one another, so you can jump to any chapter you need. However, you should read
at least Chapter 1 to familiarize yourself with some of my conventions. You will also need to understand
the material presented in Chapter 9 in order to understand Chapter 10.

Chapter 1 tells you how to install the software provided on the CD-ROM. The chapter also tells you
about some of the conventions I use and then provides you with an example on how to use some of the
modules presented in this book. I decided to include this information early in the book to allow you to
start using the code as soon as possible.

Chapter 2 introduces real-time systems concepts such as foreground/background systems, critical
sections, resources, multitasking, context switching, scheduling, nonpreemptive, task priorities, mutual
exclusion, semaphores, intertask communications, task synchronization, task coordination, interrupts,
clock ticks, etc.

Chapter 3 describes one of the building blocks shown in Figure 1.1, keyboards. Chapter 3 describes
keyboard basics and provides you with a general purpose module that can scan and decode any key-
board matrix from a 3x3 to an 8x8 key arrangement. The keyboard module can buffer keystrokes, repeat
the same key if the key is held down for a certain length of time, keep track of how long the key has
been pressed, and allow you to define multiple scan codes for each key. The code can be easily expanded to support larger keyboards.

Chapter 4 will show you how to control LED (Light Emitting Diode) displays. LED displays can
consist of discrete LEDs, seven-segment modules, or any combination of both. Chapter 4 provides you
with a module that can multiplex LEDs from a 3x3 to an 8x8 arrangement. The code can easily be
changed to accommodate larger displays.

Chapter 5 provides you with a software module that will control Character LCD Modules which are
based on the Hitachi HD44780 Dot Matrix LCD Controller & Driver chip. Character LCD (Liquid
Crystal Display) modules are display devices that can display alphanumeric data.

Chapter 6 describes a software-driven clock/calendar module that keeps track of hours, minutes, sec-
onds, days, months, years (including leap years) and day-of-week. The code also provides you with a
32-bit timestamp which can be used to mark the occurrence of events.

Chapter 7 describes a module that manages up to 250 countdown timers. Each timer can be preset to
timeout after up to 100 hours with 0.1 second resolution. You can define a function that will be executed
when the timer expires (one for each timer).

Chapter 8 provides a module that can read discrete inputs and control discrete outputs (up to 250
each). For discrete inputs, the module will tell you whether the input is high, low, transitioned from low
to high, high to low or both. When a transition is detected, a user-definable function can be executed
(one for each input). Each discrete input can also stimulate a toggle action (push-ON, push-OFF). Each
discrete output can be turned ON, turned OFF, or made to blink at a user-definable rate.

Chapter 9 will give you tools to improve the efficiency of mathematical calculations in embedded
processors. The concepts presented in this chapter will be used in Chapter 10.
Chapter 10 describes how to read and scale analog inputs and how to scale and control analog outputs. This chapter also provides you with code that will read and scale up to 250 analog inputs and scale and update up to 250 analog outputs.

Chapter 11 discusses asynchronous serial communications and specifically provides you with code that performs buffered serial I/O on a PC. There are actually two versions of this code. One version can be used by a DOS application while the other assumes the presence of a real-time kernel.

Appendix A describes how to use MicroC/OS-II. The Real-Time Kernel, μC/OS-II (for short) is a portable, ROM-able, preemptive, real-time, multitasking kernel. The internals of μC/OS-II are fully described in my other book, MicroC/OS-II, The Real-Time Kernel, which is also available (along with a diskette containing the source code) from R&D Books (see the ad at the back of the book). Most of the code presented in Embedded Systems Building Blocks assumes the presence of a real-time kernel. Specifically, I make use of semaphores and time delays which are available on most (if not all) commercially-available real-time kernels. To allow you to use the code in this book, I have included a compiled version of μC/OS-II (compiled using a Borland C++ v5.51 compiler for an Intel 80x86 Large Model).

Appendix B describes some of my programming conventions. Specifically, I describe my directory structures and C programming style.

Appendix C lists the acronyms, abbreviations, and mnemonics that I used in the code presented in this book.

Appendix D presents two DOS utilities that I use: TO and HPLIST. TO is a utility that I use to quickly move between MS-DOS directories without having to type the CD (change directory) command. HPLIST is a utility to print C source code in compressed mode (i.e., 17 CP) and allows you to specify page breaks. The printer is assumed to be to a Hewlett Packard (HP) Laserjet type printer.

Appendix E describes how to install the source code provided on the Companion CD-ROM included with this book and describes the licensing policy with regards to using the code in commercial applications.

Web Site
To provide better support to you, I created the μC/OS-II web site (http://μCOS-II.com). You can obtain information about:
• news on μC/OS, μC/OS-II, and Embedded Systems Building Blocks,
• upgrades,
• bug fixes,
• answers to frequently asked questions (FAQs),
• application notes,
• books,
• classes,
• links to other web sites, and more.
Bibliography

Gusule, Jack G.
The Art of Programming Embedded Systems
San Diego, California
Academic Press, Inc.
ISBN 0-12-274880-8

McConnell, Steve
Code Complete, A Practical Handbook of Software Construction
Redmond, Washington
Microsoft Press
ISBN 1-55615-484-4
Chapter 1

Sample Code

This chapter provides you with an example on how to use some of the embedded systems building blocks described in this book. I decided to include this chapter early in the book to allow you to start using the code as soon as possible. Before getting into the sample code, I will describe some of the conventions I use throughout the book.

The sample code was compiled using the Borland International (now called Inprise) C/C++ compiler V4.9.1 and options were selected to generate code for an Intel/AMD 80186 processor (large memory model) although the compiler was also instructed to generate floating-point instructions. I realize that the 80186 doesn't have hardware assisted but most PCs nowadays contain at least a 80486 processor which has floating-point hardware. The code was actually run and tested on a 300 MHz Intel Pentium-II based PC which can be viewed as a super fast 80186 processor (at least for my purpose). I chose a PC as my target system for a number of reasons. First and foremost, it's a lot easier to test code on a PC than on any other embedded environment (i.e., evaluation board, emulator etc.) — there are no EPROMs to burn, no downloads to EPROM emulators, CPU emulators, etc. You simply compile, link, and run. Second, the 80186 object code (Real Mode, Large Model) generated using the Borland C/C++ compiler is compatible with all 80x86 derivative processors from Intel or AMD.

Embedded Systems Building Blocks assumes the presence of a real-time kernel. For your convenience, I included a copy (in object form) of \mu\text{C/OS-II}: The Real-Time Kernel (see Appendix A for details).

1.00 Installing Embedded Systems Building Blocks

R&D Books has included a companion CD-ROM to Embedded Systems Building Blocks (ESBB). The CD-ROM is in MS-DOS format and contains all the source code provided in this book. It is assumed that you have a DOS, Windows 95, Windows 98, or Windows NT-based computer system running on an Intel, Pentium, or Pentium-II processor. You will need less than about 10 Mbytes of free disk space to install ESBB and its source files on your system.

Before starting the installation, make a backup copy of the files found on the companion CD-ROM. To install the code provided on the CD-ROM, follow these steps.
1. Load DOS (or open a DOS box in Windows 95/98/NT) and specify the C: drive as the default drive
2. Insert the companion CD-ROM in your CD drive
3. Enter `c: \> INSTALL <drive>\{drive\}`
   Note that `<drive>` is the drive letter where your CD is found and, `{drive}` is an optional drive letter indicating the destination disk on which the source code provided in this book will be installed. If you do not specify a drive, the source code will be installed on the current drive.
   INSTALL is a DOS batch file called INSTALL.bat and is found in the root directory of the companion CD-ROM. INSTALL.bat will create a `SOFTWARE` directory on the specified destination drive.
   INSTALL.bat will then change the directory to `SOFTWARE` and copy any file `SOFTWARE.exe` from the A: drive to this directory. INSTALL.bat will then execute `SOFTWARE.exe` which will create all other directories under `SOFTWARE` and transfer all source and executable files provided in this book. Upon completion, INSTALL.bat will delete `SOFTWARE.exe` and change the directory to `SOFTWARE\BLOCKS\SAMPLE\TEST` where the example code executable is found.
   Make sure you read the README file on the companion CD-ROM for last minute changes and notes.
   Also see Appendix E for a list of files and directories created.

### 1.01 How Each Chapter Is Organized

Each chapter in this book briefly introduces and describes the features of the "Embedded Systems Building Blocks" provided in the chapter. A more detailed description generally follows the introduction.

Next, I describe the internals of the module. You will find:
- the name of the directory where the module’s files are located,
- the name of the files for the building block,
- the naming conventions related to the module, and
- the step-by-step description of how the module works.

Your application interfaces with each module through functions. Interface functions allow the details of the module to be hidden from your own code. This is called data abstraction. If done properly, data abstraction allows you to change the implementation details of the module without affecting your application code. In other words, your application always sees the same module even though you may change the internals of the module. Each interface function is presented along with a description of how to use the function and what arguments are expected.

The modules provided in this book have been developed for use on fairly low-end 8-bit processors. I purchased an IBM PC/AT compatible breadboard to test some of the hardware aspects of the modules presented in this book. This breadboard made testing a breeze. The breadboard I used was the JDR Microdevices (see bibliography) PDS-601 which cost only $80. The PDS-601 contains an 80S bus interface, decoding logic, an Intel 8255A chip, an Intel 8253 (similar to an 8255), and a large breadboard area.

In every building block, I tried to include target-specific code into a few functions and configuration constants, i.e., #defines. This allows you to easily adapt the code to your own environment. Thus, each chapter has a configuration section which describes how to change the code so that it can work in your target system.

Some of the chapters, specifically Chapters 3, 4, 8, 10 and 11, include a section called, "How to Use the ??? Module." This section provides an example on how you can actually use the module in an appli-
Chapter 1: Sample Code — 3
cation. The example describes how to properly initialize the code and how to invoke some of its services.

Each chapter ends with a bibliography, source code listings, and pointers to one or more data sheets (stored on the CD-ROM) of an electronic components mentioned in the chapter.

### 1.02 INCLUDES.H

You will notice that every .c file in this book contains the following declaration:

```c
#include "includes.h"
```

`includes.h` allows every .c file in your project to be written without concern about which header file will actually be included. In other words, `includes.h` is a master include file. The only drawback is that `includes.h` includes header files that are not pertinent to some of the .c file being compiled. This means that each file will require extra time to compile. This inconvenience is offset by code portability. You can certainly edit `includes.h` to add your own header files. The actual `includes.h` I used is found in Listing 1.24 at the end of this chapter.

### 1.03 Compiler Independent Data Types

Because different microprocessors have different word lengths, I have created a number of type definitions that ensure portability (see SOFTWARE/UCOS-II/drivers/PP-OS_CPU.h (see Appendix A, Listing A.1) for the 80386 real-mode, large model). Specifically, ESBB and UCOS-II code never make use of C's `short`, `int`, and `long` data types because they are inherently non-portable. Instead, I defined integer data types that are both portable and intuitive as shown below.

```c
typedef unsigned char ROOLAN;
typedef unsigned char INT8U;
typedef signed char INT8S;
typedef unsigned int INT16U;
typedef signed int INT16S;
typedef unsigned long INT32U;
typedef signed long INT32S;
typedef float FP32;
typedef double FP64;
```

The `INT16U` data type, for example, always represents a 16-bit unsigned integer. ESBB, UCOS-II, and your application code can now assume that the range of values for variables declared with this type is from 0 to 65535. A compiler for a 32-bit processor could specify that an `INT16U` would be declared as an `unsigned short` instead of an `unsigned int`. Where the code is concerned, however, it still

---

Footnotes:
deals with an INTL file. The above code fragment provide the declarations for the 80x86 and the Borland C/C++ compiler as an example.

1.04 CFG.C and CFG.H

To allow you to easily adapt the code in this book to your environment, I created two user-configurable files called CFG.C and CFG.H. All the target-specific code has been conveniently located for you in CFG.C and CFG.H. You don’t have to edit every .C and .H file to use the code in this book. If you adapt CFG.C and CFG.H to your environment, you can use every module “as is”.

CFG.C (Listing 1.23) contains the hardware-specific functions of the modules presented in this book. CFG.H (Listing 1.23) contains the configuration #defines for each module. CFG.C and CFG.H are found in the \SOFTWARE\BLOCKS\SAMPLE\SOURCE\directory. In order to use CFG.C and CFG.H, you must “tell” the compiler to ignore the same declarations in the code for the modules. You accomplish this by defining the constants CFG.C and CFG.H in INCLUDES.H.

1.05 Global Variables

The following is a technique that I use to declare global variables. As you know, a global variable needs to be allocated storage space in RAM and must be referenced by other modules using the C keyword extern. Declarations must thus be placed in both the .C and the .H files. This duplication of declarations, however, can lead to mistakes. The technique described in this section only requires a single declaration in the header file, but is a little tricky to understand. However, once you know how this technique works, you will apply it mechanically.

In all .H files that define global variables, you will find the following declaration:

Listing 1.3 External references

```c
#include xxxGLOBALS
#define xxx_EXT

#define xxx_EXT extern
#endif
```

Each variable that needs to be declared global will be prefixed with xxx_EXT in the .H file. “xxx” represents a prefix identifying the module name. The module’s .C file will contain the following declaration:

Listing 1.4 .C file declarations of global variables

```c
#define xxxGLOBALS
#include "includes.h"
```

When the compiler processes the .C file it forces xxx_EXT (found in the corresponding .H file) to “nothing” (because xxxGLOBALS is defined) and thus each global variable will be allocated storage space. When the compiler processes the other .C files, xxxGLOBALS will not be defined and thus
Chapter 1: Sample Code — 5

XEQ_EXT will be set to extern, allowing you to reference the global variable. To illustrate the concept, let’s look at DIO.H (from Chapter 8) which contains the following declarations:

**Listing 1.5 Example using DIO.H**

```c
#ifdef DIO_GLOBALS
#define DIO_EXT
#endif
#define DIO_EXT extern
#endif

DIO_EXT DIO_D1 DETH1[DIO_MAX_D1];
DIO_EXT DIO_D0 DOTB1[DIO_MAX_D0];
```

DIO.C contains the following declarations:

**Listing 1.6 Example using DIO.C**

```c
#define DIO_GLOBALS
#include "includes.h"
```

When the compiler processes DIO.C, it makes the header file (DIO.H) appear as shown below because DIO_EXT is set to "nothing":

**Listing 1.7 Expanding DIO.H**

```c
DIO_D1 DETH1[DIO_MAX_D1];
DIO_D0 DOTB1[DIO_MAX_D0];
```

The compiler is thus told to allocate storage for these variables. When the compiler processes any other .C files, the header file (DIO.H) looks as shown by the following code because DIO_GLOBALS is not defined and thus DIO_EXT is set to extern.

**Listing 1.8 Expanded .H file other than DIO.H**

```c
extern DIO_D1 DETH1[DIO_MAX_D1];
extern DIO_D0 DOTB1[DIO_MAX_D0];
```

In this case, no storage is allocated and any .C file can access these variables. The nice thing about this technique is that the declaration for the variables is done in only one file, the .H file.
1.06 OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()

Throughout the source code provided in this book, you will see calls to the following macros:

**OS_ENTER_CRITICAL()** and **OS_EXIT_CRITICAL()**. **OS_ENTER_CRITICAL()** is a macro that disables interrupts and **OS_EXIT_CRITICAL()** is a macro that enables interrupts. Disabling and enabling interrupts is done to protect critical sections of code. These macros are obviously processor specific and are different for each processor. These macros are found in OS_CPU_H (see Appendix A, Listing A.1) and for the code provided in this book, these macros are defined as follows.

```c
#define OS_ENTER_CRITICAL() asm ("PUSH: CLC")
#define OS_EXIT_CRITICAL() asm ("POP F"
```

Your application code can make use of these macros as long as you realize that they are used to disable and enable interrupts. Disabling interrupts obviously affects interrupt latency so be careful. You can also protect critical sections using semaphores.

1.07 ESBB Sample Code

The sample code is found in the `SOFTWARE/BLOCKS/SAMPLE/SOURCE` of the installation directory. This source directory contains the following files:

- **CFG.C** (Listing 1.22)
- **CFG.H** (Listing 1.23)
- **INCLUDES.H** (Listing 1.24)
- **OS_CPU.H** (Listing 1.26)
- **TEST.C** (Listing 1.27)
- **TEST.LIBC** (Listing 1.28)

**CFG.C** and **CFG.H** were discussed in section 1.04. **INCLUDES.H** was discussed in section 1.02. **OS_CPU.H** is a configuration file needed by µCOS-II and should not be altered unless you obtain the full source version of µCOS-II (see Appendix A for details). **TEST.LIBC** is the linker command file and is shown in Listing 1.28.

The sample code is actually found in **TEST.C** (see Listing 1.27) and will be described in this section. The sample provided (along with the building blocks used) in this chapter was compiled using the Borland C/C++ V4.51 compiler in a DOS box on a Windows 95 platform. To make the process easy, I created a makefile called **TEST.MAK** (see Listing 1.29). The makefile is invoked by the batch file **MAKETEST.BAT** (see Listing 1.25). Both files are found in the `SOFTWARE/BLOCKS/SAMPLE/TEST` directory. To build the sample code, you need to change your current directory (using the DOS CD command) to `SOFTWARE/BLOCKS/SAMPLE/TEST` and type:

```
C:\SOFTWARE/BLOCKS/SAMPLE/TEST > MAKETEST
```
You should note that my Borland compiler is installed on my E: drive, but you can easily change the makfile to have it point to the proper directory and drive by changing the following lines in TEST.MAK:

[Listing 1.10 Tool declarations in TEST.MAK]

```
# tool declarations

BORLAND=E:\BCC
BORLAND_ED=E:\BCC\BCS\BIN
```

μC/OS-II is a scalable operating system which means that the code size of μC/OS-II can be reduced if you are not using all of its services. However, because μC/OS-II is not provided in source form in this book, you will be limited to the features I needed to run the sample code. You can obtain the full source version of μC/OS-II by obtaining a copy of my other book, MicroC/OS-II, The Real-Time Kernel, ISBN 0-97930-543-6.

Once built, you can run the sample code by typing:

```
C:\SOFTWARE\BLOCKS\SAMPLE\TEST > TEST
```

The display on your PC should look as shown in Figure 1.1. You will notice that there is no sample code for Chapter 3 “Keyboards”, Chapter 4 “Multiplexed LED Displays”, and Chapter 5 “Character LCD Modules” because you would need some special hardware which I didn’t want to assume.
Figure 1.1  **DOS Window display for Sample code**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>#Tasks: 14</th>
<th>#Task switch/sec: 345</th>
<th>CPU Usage: 1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 8, Discrete I/O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 9, Keyboards</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 6, Time-Of-Day Clock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 10, Analog I/O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date: Friday December 31, 1999</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time: 23:58:00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TS: 1999-12-31 23:58:00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date: 11 uS Time: 4 uS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 7, Timer Manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task: 01:03:0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task: 00:00:0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sample code basically consists of 13 tasks as listed in Table 1.1.

### Table 1.1  **Tasks in sample code**

<table>
<thead>
<tr>
<th>Module/File</th>
<th>Task</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST.C</td>
<td>Analog I/O Test Task</td>
<td>10 (Highest)</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Clock Test Task</td>
<td>11</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Asynchronous Serial Comm. Tx Test Task</td>
<td>12</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Asynchronous Serial Comm. Rx Test Task</td>
<td>13</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Discrete I/O Test Task</td>
<td>14</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Timer Manager Test Task</td>
<td>15</td>
</tr>
<tr>
<td>TEST.C</td>
<td>Statistic / PC Keyboard Test Task</td>
<td>16</td>
</tr>
<tr>
<td>CLK.C</td>
<td>Time-of-Day Clock Task</td>
<td>51</td>
</tr>
<tr>
<td>TM.R.C</td>
<td>Timer Manager Task</td>
<td>52</td>
</tr>
<tr>
<td>DIO.C</td>
<td>Discrete I/O Manager Task</td>
<td>53</td>
</tr>
<tr>
<td>µCOS-II</td>
<td>Analog I/O Manager Task</td>
<td>54</td>
</tr>
<tr>
<td>µCOS-II</td>
<td>Statistic Task</td>
<td>62</td>
</tr>
<tr>
<td>µCOS-II</td>
<td>Idle Task</td>
<td>63 (Lowest)</td>
</tr>
</tbody>
</table>
μC/OS-II creates two internal tasks; the idle task and a task that determines CPU usage. Four of the building blocks each create a task and TEST.C creates the other 7 tasks.

As can be seen from the screen of Figure 1.1, there is no sample code for Chapters 3, 4, and 5 because they would require hardware not available on a regular PC.

For Chapter 6, the test code sets up the CLK module's current date and time to December 31, 1999 at 11:58 PM to show you that the CLK module is year 2000 (Y2K) compliant by correctly rolling over to Saturday, January 1, 2000 in two minutes. However, by the time you get this book, the Y2K problem should be a thing of the past. You should note that the CLK module does not change the actual date and time of your PC. When you run the code, you will also see the timestamp being updated. Also, I used the elapsed time measurement functions in PC.C to determine the execution time of CLKtime()and CLKformatTime() functions.

The sample code for Chapter 7 sets up 2 timers. The first timer expires after 1 minute and 3 seconds and the second expires after 2 minutes. When the first timer expires, the message "Timer #1 Timed Out!" will be displayed just below the line showing time #0. When the second timer expires, the message, "Timer #1 Timed Out!" below its timer. Instead of displaying messages, you could perform any other operation including signaling a task.

For Chapter 8, although the DIO task continuously reads discrete inputs (DI), I don't actually make use of that feature because it would require external hardware. Instead, I only set up 3 discrete outputs (DO) for which I display the state of these outputs on the screen (TRUE or FALSE for DO #0, TRUE or FALSE for DO #1 and, FALSE (99) for DO #2). The first discrete output is set up to produce a 'blinking' output with a 50% duty-cycle (50% ON, 50% OFF) at a rate of 1 Hz. The second discrete output is also set up to 'blink' but does so at half the rate of the first channel (0.5 Hz). Finally, the third output blinks with a 25% duty cycle but runs in 'asynchronous mode' (see Chapter 8).

There is no sample code provided for Chapter 9 because this chapter does not actually contain a building block.

For Chapter 10, instead of having you work with an ADC on a PC, I simply decided to 'simulate' the reading of an analog input which increases by 10 counts every time an ADC reading is required. When the counts reach 32700 (assuming a simulated 15-bit ADC), the counts are reset back to 0. Note that there aren't too many commercial 15-bit ADCs, but you will see in Chapter 10, you can fake your software into thinking that all ADCs with less than 16 bits can actually look like they have 15 bits.

For Chapter 11, I created two tasks. One task sends the value of a counter to the other task. However, this message is actually sent through the serial port (COM1 on the PC). To see the operation of the sample code, you'll need to truly run in DOS (i.e., not in a DOS box under Windows 95/98 or NT) and connect the Tx and Rx lines of COM1 on your PC together. In order to accomplish this, I used a 'Laplink' serial cable (you can buy this at any good computer store) that I plugged into my PC. I then shorted pins 2 and 3 of either the DB-9 female or DB25 female connector using a paper clip.

1.07.01 main() A μC/OS-II application looks just like any other DOS application. You compile and link your code just as if you would do a single threaded application running under DOS. The .exe file that you create is loaded and executed by DOS, and execution of your applications starts from main().

The sample code (SLEEP1220.C) serves two purposes. First, if you invoke the sample code from the DOS prompt and specify either "display" or "Display" ([L:.111]) as an argument, your screen will display the corresponding character that corresponds to each byte value from 0x00 to 0xFF. In other words, to see the characters mapping simply type:

TEST display
or,

```
TEST DISPLAY
```
at the DOS prompt.

If you simply typed `TEST` at the DOS prompt, then `main()` clears the screen to ensure we don’t have any characters left over from the previous DOS session [L1.11(2)]. Note that I specified to use white letters on a black background. Since the screen will be cleared, I could have simply specified to use a black background and not specify a foreground. If I did this, and you decided to return to DOS then you would not see anything on the screen! It’s always better to specify a visible foreground just for this reason.

**Listing 1.11 main()**

```c
void main (int argc, char *argv[])
{
    if (argc > 1) {
        if (strncmp(argv[1], "display", 6) == 0) {
            TestDispMap();
        }
        exit(0);
    }

    PC_DispColScr(DISP_FND_WHITE + DISP_NOND_BLACK);
    OSInit();
    OSSPInit();
    PC_DOSSaveReturn();
    PC_VectSet(uCOS, OCMxSw);
    OSTaskCreateEx(ThpstTask, *0.10,
        #TestStartTaskStk(TASK_STK_SIZE,
                        START_TASK_PRIO,
                        START_TASK_PRIO,
                    #TestStartTaskStk[0],
                    TASK_STK_SIZE,
                (void *)0,
                OS_TASK_OPT_SAVESPPi);
    OStartAll();
}
```

A requirement of uCOS-II is that you call `OSInit()` [L1.11(3)] before you invoke any of its other services. `OSInit()` creates two tasks: an idle task which executes when no other task is ready-to-run and a statistic task which consumes CPU usage.
Because the code is assumed to run on a 80486 or Pentium class computer, I decided to make use of hardware assisted floating-point and thus, we need to invoke the code that will tell µCOS-II to initialize the floating-point support [L.11.6].

The current DOS environment is then saved by calling PC_DOSSaveReturn() [L.11.3]. This allows us to switch to DOS as we had never started µCOS-II. A lot happens in PC_DOSSaveReturn() and this is all explained in Chapter 12 (sections 12.01).

main() then calls µC_SVecSet() [L.11.6(3)] to install µCOS-II’s context switch handler. Task context switching is done by issuing an 80386 INT 13 instruction to this vector location. I decided to use vector 0x80 (i.e., 128) because it’s not used by either DOS or the BDIS.

Before starting multitasking, I create one task [L.11.7(7)] called TestStatTask(). It is very important that you create at least one task before multitasking begins with QStart() [L.11.8(8)]. Failure to do this will certainly make your application crash. Once QStart() is called, multitasking begins and µCOS-II runs the highest priority task that is ready to run. This happens to be TestStatTask() which will be described next.

1.07.02 TestStatTask()

Initialization of the sample code continues in TestStatTask(). µCOS-II needs a little more setup which is accomplished by “installing” the tick handler [L.12.13]. Next, I decided to change the tick rate from the default DOS 18.2 Hz to 200 Hz [L.11.12(2)]. This allows better granularity when we need to run tasks at regular intervals. You should note that a lot of setup has to be done to move from the DOS environment to the µCOS-II environment. In an actual embedded system, there would be no need to save the CPU registers to return back to DOS (see µC_DOSSaveReturn() because we would most likely not return back to DOS to begin with. We would, however, most likely need to install the tick ISR handler and set a hardware timer which would provide a tick source.

Note that main() purposely didn’t set the interrupt vector to µCOS-II’s tick handler because you don’t want a tick interrupt to occur before the operating system (µCOS-II) is fully initialized and running. If you run code in an embedded application, you should always enable the tick (as I have done here) from within the first task.

Before we create any other tasks, we need to determine how fast you particular FC is. This is done by calling the µCOS-II function QStartInit() [L.12.4]. Calling QStartInit() allows µCOS-II to determine the CPU usage (in percent) of your CPU while your application (in this case, the test code) is running.

Once µCOS-II knows about your CPU, we call TestInitModules() to initialize the building blocks that are used in the sample code. The code for TestInitModules() is shown in Listing 1.13.

Listing 1.12 Beginning of TestStatTask()

```c
void TestStatTask(void *pdata) {

    INPUT i;

    DEFERD key;

    char s[51];

    pdata = pdata;

    ...
```
Listing 1.12  Beginning of TestStartTask()

OS_ENTER_CRITICAL();
PC_VectSet(G588, OSTickISR);
PC_SetTickRate(OS_TICKS_PER_SEC);
OS_ENTER_CRITICAL();

PC_DispStr(0, 22, "Determining CPU's capacity ...");
DISP_FUND_WHITE;
OSStartInit();
PC_DispClrRow(22, DISP_FUND_WHITE + DISP_BLACK);
TestInitModules();

TestInitModules() starts off by initializing the elapsed time measurement provided in the PC services (see Chapter 2 [L.1.13(2)]). Because MODULE_KEY_BAT [L.1.13(2)], and MODULE_LCD [L.1.13(3)] and MODULE_LCD [L.1.13(3)] are set to 0 in INCLUDES.h, the keyboard, LED, and LCD building blocks are not initialized. All of the other building blocks, however, are initialized because they are enabled in INCLUDES.h [L.1.13(3-8)]. The last building block CC3000 uses the RTOS version (see Chapter 11) because it is used in conjunction with μCOS-II. In this case, I assume that CC3000 on your PC is used for the test and it is set up to communicate at 9600 baud [L.1.13(10-13)].

Listing 1.13 is part of TestStartTask() and is responsible for creating the test tasks which will exercise the building blocks used in the sample code. Each task that is to be managed by μCOS-II must be created. This allows μCOS-II to know about where the task code resides, what stack is to be allocated to the task, what priority is given to the task, and more. You can find out more about OSTask_CreateTask() in Appendix A.

Listing 1.13  TestInitModules()

static void TestInitModules(void)
{  (1)
    PC_ElapsedInit();
    //if MODULE_KEY_BAT
    KeyInit();
    #endif
    //if MODULE_LCD
    DispInit();
    #endif
}
Listing 1.13  TestInitModules()

```c
#if MODULE_LCD
  DspInit(4, 30);
#endif

#if MODULE_CLK
  ClkInit();
#endif

#if MODULE_TMR
  TmrInit();
#endif

#if MODULE_LED
  LEDInit();
#endif

#if MODULE_ASO
  ASOInit();
#endif

#if MODULE_COMM_BOND
  CommInit();
#endif

#if MODULE_COMM_P700
  CommInit();
#endif

#if MODULE_COMM_PC
  CommCfgPort(COM1, 9600, 8, COMM_PARITY_NONE, 11);
  CommSetIntVec(COM1);
  CommRxIntEn(COM1);
#endif
```
Listing 1.14  Creation of test tasks (TestStatTask())

```c
void *TestStatTask()
{
    TaskStat ticks[8];
    int i;

    for (i = 0; i < 8; i++)
    {
        ticks[i] = 0;
    }

    while (1)
    {
        TaskStat taskstat[8];

        for (i = 0; i < 8; i++)
        {
            taskstat[i] = 0;
        }

        for (i = 0; i < 8; i++)
        {
            ticks[i] = taskstat[i];
        }
    }
}
```

Listing 1.14 Creation of test tasks (TestStatTask)

```c
testStatTask,
(void *)0,
&TestStatTaskStk[TASK_STK_SIZE],
TEST_DIO_TASK_PRIO, TEST_DIO_TASK_PRIO,
&TestDIOtaskStk[0],
TASK_STK_SIZE,
(void *)0,
OS_TASK_OPT_SAV_RTFP);

GTaskCreateTst(TstADTask,
(void *)0,
&TestADTaskStk[TASK_STK_SIZE],
TEST_AD_TASK_PRIO, TEST_AD_TASK_PRIO,
&TestADTaskStk[0],
TASK_STK_SIZE,
(void *)0,
OS_TASK_OPT_SAV_RTFP);
```

Listing 1.15 is also part of TestStatTask(). The literals (i.e., text that doesn’t change on the screen) are displayed by calling TestDisplay() [L.1.15(1)]. This is done to avoid wasting CPU time updating the display with information that doesn’t change. Next TestStatTask() displays the current version of µC/OS-III at the bottom left hand corner of the screen [L.1.15(2)].

TestStatTask() then enters an infinite loop. This is the main body of the task code. Every second (you’ll see why later) the following information is displayed at the bottom of the screen:

- the number of tasks created (GTaskCnt) [L.1.15(3)],
- the number of context switches (i.e., task switches) per second (GCtxSwCnt) [L.1.15(4)] and,
- the percentage of the CPU used by the sample code (GCPUUsage) [L.1.15(5)].

You may wonder why I am updating the display of the task counter every second since there are no other tasks created from here on. The reason is to allow you to create other tasks which could be delayed. In other words, you may decide to create a task only after some time has elapsed.

This task then checks to see if a key has been pressed [L.1.15(6)] and if so, determines whether the key pressed was the Esc key [L.1.15(7)]. If the Esc key is pressed, the sample code exits back to DOS. Before we can return to DOS, though, we must instantiate the original DOS context ISR vector [L.1.15(8)]. Returning back to DOS is accomplished by calling SC_DOSReturn() [L.1.15(9)] (see Chapter 12, section 12.01).

In order to display the number of context switches per second, the global variable GCtxSwCnt must be cleared every second [L.1.15(10)].

To prevent this task from using all the CPU (remember that we are in an infinite loop), the task calls the µCOS-II service GOSTime00HREM() [L.1.15(11)] (see Appendix A). This call suspends the current task until some time expires. In one case, the arguments 0, 0, 1, 0 specify 1 second delay. When one second expires, µCOS-II will resume execution of this task immediately after the call to GOSTime00HREM() or, at the top of the for t loop.
Listing 1.15 Task portion of TestStatTask()

```c
TestStatTask();

sprintf(s, "\%d\%d\%d", OSVersion() / 100, OSVersion() % 100);
PC_DispStr(13, 23, a, DISP_FONDO_YELLOW + DISP_FONDO_BLUE);

for (;;) {
    printf(s, "\%d", OSHasActive);
    PC_DispStr(30, 23, a, DISP_FONDO_BLUE + DISP_FONDO_CYAN);
    printf(s, "\%d", OSKbdBusy);
    PC_DispStr(56, 23, a, DISP_FONDO_BLUE + DISP_FONDO_CYAN);
    printf(s, "\%d", OSKbdImage);
    PC_DispStr(75, 23, a, DISP_FONDO_BLUE + DISP_FONDO_CYAN);
    if (PC_GetKey(&key) == TRUE) {
        if (key == 0x1B) {
#if MODULE_COMPC
            ComPCIntEnVest(ComPC);
#endif
            PC_D3GReturn();
        } else {
            OSGetKeyWord(0, 0, 0, 0);
        }
    }
}
```

1.07.03 TestClikTask()

TestClikTask() is shown in Listing 1.16 and this task shows some of the functions of the Clik building block of Chapter 6 which consist of code to maintain a time-of-day clock.

We first set up the current time-of-day and date to December 31, 1999 at 12:58 PM (i.e., 2 minutes before midnight) [L.1.16(1)].

The task portion of the code (i.e., the infinite loop) is then entered and the function PC_ElapsedStartTime() is invoked [L.1.16(2)] to setup the PC’s timer #2 so that it can be used to measure the execution time of ClikFormatDate() [L.1.16(3)]. ClikFormatDate() formats the cur-
Listing 1.16 TestClkTask()  

```c
void TestClkTask(void *data) {
    char s[81];
    INT16U time;
    TS ts;

    data = data;

    CLKSetDateTime(12, 31, 1999, 23, 58, 0);
    (1)

    for (;;) {
        PC_ElapsedStart();
        CLKFormatDate(2, s);
        (2)
        time = PC_ElapsedStop();
        (3)
        PC_DisPlayStr( 8, 11, " ", DISP_PND_MIDBLUE);
        PC_DisPlayStr( 8, 11, s, DISP_PND_BLUE + DISP_PND_CYAN);
        printf(1, "%d ms", time);
        (4)

        PC_ElapsedStart();
        CLKFormatTime(1, s);
        (5)
        time = PC_ElapsedStop();
        (6)
        PC_DisPlayStr( 8, 12, s, DISP_PND_BLUE + DISP_PND_CYAN);
        printf(1, "%d ms", time);
        (7)

        PC_DisPlayStr(22, 14, s, DISP_PND_RED + DISP_PND_LIGHT_GRAY);
        (8)
        ts = CLKGetTS();
        CLKFormatTS(2, ts, s);
        (9)
        PC_DisPlayStr( 8, 13, s, DISP_PND_BLUE + DISP_PND_CYAN);
        (10)
        ODTimeDelayNSM(0, 0, 0, 100);
    }  
}
```

The current date maintained by the CLK building block is an ASCII string. The format selected (i.e., 2) is "Day Month DD YYYY" where 'Day' is the day of the week (Monday, Tuesday ...), 'Month' is the month of the year (January, February ...), 'DD' is the calendar day (1, 2, 3 ...), and 'YYYY' is the current year using 4 digits. The execution time of CLKFormatDate() is captured by calling PC_ElapsedStop(); [L1.16(4)] which returns the time in microseconds. Both the current date and the execution time are then displayed.
The function PC_ElapsedStart() is called again [L.1.16(5)] to setup the PC's timer #2 so that it can be used to measure the execution time of CLKFormatTime() [L.1.16(6)]. CLKFormatTime() formats the current time maintained by the CLK building block into an ASCII string. The format selected (i.e., 1) is "HHh:MMm:SSs" which consist of the current time in 24 hour format (i.e., up to 23:59:59). The execution time of CLKFormatTime() is captured by also calling PC_ElapsedStop() [L.1.16(7)]. Both the current time and the execution time are then displayed.

The CLK building block also maintains a special format called a timestamp. A timestamp basically captures the date and time in a single 32-bit variable as shown in Figure 1.2. This allows your application to mark an event such as the occurrence of an error or the reception of a message and capture when that event occurred. You can thus obtain the current timestamp by calling CLKGetTS() [L.1.16(8)]. It is easier to display the timestamp in ASCII which is why CLKFormatTS() is invoked [L.1.16(9)]. The format selected "YYYY-MM-DD HH:MM:SS" is new to this second edition. I personally like this format because it displays the year as 4 digits followed by the month and then the day. What's also convenient about the ASCII format is that it can be sorted easily. OSGetDlyTIME() is then called to suspend this task for 100 milliseconds. In other words, this task executes 10 times per second [L.1.16(10)].

**Figure 1.2 Timestamp format**

<table>
<thead>
<tr>
<th>Year</th>
<th>Month</th>
<th>Day</th>
<th>Hours</th>
<th>Minutes</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2023-01-01</td>
<td>2022-06-17</td>
<td>21-01-01</td>
<td>01-12</td>
<td>0.31</td>
<td>0.59</td>
</tr>
</tbody>
</table>

1.07.04 TestTimerTask()

TestTimerTask() is shown in Listing 1.17 and shows some of the functions of the TIM building block of Chapter 7 which consists of code that maintains up to 250 down counters that can be set to any time from 1 tenth of a second to 99 minutes, 59 seconds and 9 tenths of a second or 99.599 (using the nonstandard macro). When a timer expires, it can optionally call a user-definable function.

We first start up by configuring timer #0's timeout function [L.1.17(1)]. When timer #0 times out, it will call TestTimer() which simply displays "Timer #0 Timed Out!". The timer is then initialized to 1.039 [L.1.17(2)] and then it's started [L.1.17(3)].
We then configure a second timer, timer #2's timeout function (L1.17(4)). When timer #1 times out, it will call TestTask2() which also displays a similar message, "Timer #2 Timed Out". The timer is then initialized to 2000 (L1.17(5)) and then it is started (L1.17(6)).

**Listing 1.17 TestThurTask()**

```c
void TestThurTask (void *data)
{
    Chat s[48];
    INIT(1) time;

    data = data;
    TestCtgRectG, TestCtgRectO, (void *)0;
    TestCtgMenu(1, 2, 9);
    TestStart(0);
    TestCtgRectL, TestCtgRectN, (void *)0;
    TestCtgMenu(1, 2, 0, 0);
    TestStart(1);

    for (;;) {
        TestFormat(0, e);
        PC_DispStr(9, 16, a, DISP_LEFT_RED+DISP_LEFT_GRAY);
        TestFormat(1, 0);
        PC_DispStr(9, 10, a, DISP_GRAY_RED+DISP_GRAY_GRAY);
        OptionalSysMem(0, 0, 0, 50);
    }
}
```

The time remaining for both timers is displayed (L1.17(7-8)) and the task body continuously loops 20 times per second (it doesn’t really need to be this fast, though) (L1.17(9)).

**1.07.05 TestDIOTask()**

TestDIOTask() is shown in Listing 1.18 and shows some of the functions of the DIO building block of Chapter 8. The DIO module reads and update up to 256 discrete inputs and outputs. A discrete input normally represents the state of an external switch (a pushbutton switch, a pressure switch, a temperature switch, etc.). A discrete output generally consists of a simple relay output to control a single light, a valve, a motor, etc.
Although the DIO task can read discrete inputs (DI), I don't actually make use of that feature because it would require external hardware. Instead, I only set up 3 discrete outputs (DO) for which I display the state of these outputs on the screen:

- For DO #0 we will display TRUE or FALSE
- For DO #1 we will display HIGH or LOW
- For DO #2 we will display ON or OFF

The DIO task [DIO_Task()] (see Chapter 8) which is responsible for updating the DIS and DIOs will execute 10 times per second (see CPULK_DIO_TASK_DELAY_TICS). To get a 10 second synchronous count value, we call DOGetSyncCtrMax() [L1.18(1)] which sets DOSyncCtrMax to 1000 (100 * 10 sec). Note that you wouldn't need to invoke this function if you didn't use the 'synchronous' mode of the DIO module.

I then configure DO #0 to blink at a rate of 1 Hz with a 50% duty cycle [L1.18(2)]. The values specified as arguments to DIOConfigBlink() do not correspond to RTOS ticks but instead, they correspond to number of updates of the DIO module. In other words, if the DIO task is updated 10 times per second — then a value of 10 represents 1 second, a value of 20 represents 2 seconds, etc. To finalize the configuration of DO #0, I need to set the mode to asynchronous blinking and non-invert the output (see Chapter 8, Figure 8.9) [L1.18(3)]. Configuration of DO #1 is similar to DO #0 except that I set the blink at a rate to 0.5 Hz (i.e., 2 seconds) [L1.18(4)]. DO #1 is also set to asynchronous blinking and non-invert the output [L1.18(5)]. Configuration of DO #2 is set to synchronous blinking and its output is also non-inverted [L1.18(6-7)].

We then enter the task body which simply obtains the state of each discrete output and displays it on the screen. This happens 10 times per second although this doesn't need to be done this fast considering that none of the outputs change this quickly.

**Listing 1.18 TestDIOTask()**

```c
void TestDIOTask (void *data) {
    BOOL state;
    data = data;
    DIOSetSyncCtrMax(100);            // (1)
    DIOConfigBlink(0, DIO_BLINK_ON, 5, 10); // (2)
    DIOConfigMode(0, DIO_MODE_BLINKASYNC, FALSE); // (3)
    DIOConfigBlink(1, DIO_BLINK_ON, 15, 20); // (4)
    DIOConfigMode(1, DIO_MODE_BLINKASYNC, FALSE); // (5)
    DIOConfigBlink(2, DIO_BLINK_ON, 25, 0); // (6)
    DIOConfigMode(2, DIO_MODE_BLINKSYNC, FALSE); // (7)
}
```
Listing 1.18  TestD10Task()

```c
for (;;) {
    state = DOGet(0);
    if (state == TRUE) {
        PC_DispStr(49, 6, "TRUE ",
                    Disp_Font_Yellow + Disp_Font_Bold);
    } else {
        PC_DispStr(49, 6, "FALSE",
                    Disp_Font_Yellow + Disp_Font_Bold);
    }
    state = DOGet(1);
    if (state == TRUE) {
        PC_DispStr(49, 7, "HIGH",
                    Disp_Font_Yellow + Disp_Font_Bold);
    } else {
        PC_DispStr(49, 7, "LOW ",
                    Disp_Font_Yellow + Disp_Font_Bold);
    }
    state = DOGet(2);
    if (state == TRUE) {
        PC_DispStr(49, 8, "ON ",
                    Disp_Font_Yellow + Disp_Font_Bold);
    } else {
        PC_DispStr(49, 8, "OFF ",
                    Disp_Font_Yellow + Disp_Font_Bold);
    }
    CoutToOlyVscr(0, 2, 0, 100);
}
```

1.07.06 TestAIOTask()

TestAIOTask() is shown in Listing 1.19 and shows some of the functions of the AIO building block of Chapter 10. The AIO module reads and updates up to 256 analog inputs and outputs. Each analog input can be configured to read just about any type of sensor (temperature, pressure, position, flow, etc.). An analog output can be made to control a large number of devices such as a valve, an actuator, a positioner, etc.

It's difficult to show the operation of this building block without actually having an ADC (Analog to Digital Converter) and a DAC (Digital to Analog Converter) on a PC. What I decided to do is simply simulate a ramping ADC and convert the value to some engineering units. I thought of using the LM-34A (see Chapter 10, Figure 10.7) as my 'simulated' sensor and generate temperatures from -50 to
about 300 degrees Fahrenheit. I assumed that my ADC would be made to look like a 16-bit signed ADC, referenced at 10 volts, the gain would be set to 2.5 volt, I have a 1.25 volt offset so that I could read negative temperatures. From Equations 10.9 and 10.10, I obtain a gain of 0.01220740 and an offset of -0.095.875 and I configure AI @ accordingly [L1.19(1)].

The task code simply counts of reading the current engineering value (i.e., the temperature of the simulated LM34A) from the analog channel [L1.19(2)] and displaying it on the screen. You should note that I didn’t need to display decimal places and thus, I converted the temperature to an integer.

The task code repeats 100 times per second [L1.19(3)]. Again, this rate is not necessary and has been chosen simply to make the CPU busy.

Listing 1.19  TestAIOTask()

```c
void TestAIOTask (void *data)
{
    char *s[81],
    PFI2 value;
    INT16S temp;
    IN64  err;

    data = data;

    ADCEnv(0, 0.0122074, -0.095, 10);
    ADCeCal(0, 1.0, 0.0);

    for (;;)
    {
        err = ADCen(0, &value);
        temp = (INT16S)value;
        sprintf(s, "%6d", temp);
        PC.DISpStr(49, 11, s, DISP_BOLD_YELLOW + DISP万台 BLUE);
        OTextedlyNum(0, 0, 0, 10);
    }
}
```

1.07.07  TestTxTask() and TestRxTask()

It is assumed that you would connect a ‘LagLink’ serial cable on COM1 and short the Tx line (pin #3) to the Rx line (pin #2) on the free end of the DB9F connector.

The TestTxTask() is shown in Listing 1.20 and shows some of the functions of the COM building-block of Chapter 11. This task simply increments a 16-bit counter, converts it to ASCII [L1.20(1)] and sends the string on COM1 one character after the other [L1.20(2)]. A delay of 5 ticks is added in case you run this code under Windows 95/98 or NT [L1.20(3)]. This is needed to accommodate overhead imposed by Windows. If you were to run this code either in DOS or on an actual embedded system, you
would not need the delay. I actually tested this code on a QNX-based machine all the way to 3800
based for a few hours without any glitches, however, it crashes with Windows 95/98.

TestTxTask() is shown in Listing 1.21 and is basically the receiving task for the transmitted mes-
sages from TestTxTask(). This task waits for characters to be received on COM1 (L1.21(1)). As each
character is received, it is placed in a buffer [L1.21(2)]. When the carriage return character (‘\n’ or
0x0D) is received, the string is terminated [L1.21(3)] and the string received is displayed [L1.21(4)]. Of
course both the transmitted and received messages should match.

Listing 1.20 TestTxTask()

```c
void TestTxTask (void *data)
{
    char  *ps;
    char   n[81];
    char ctr;

    data = data;
    ctr = 0;
    for (i=0;
        sprintf (n, "\033\n", ctr);  \(1\)
        SC_Display (49, 16, s, DISP_FUND_YELLOW + DISP_BOLD_BLUE);
        ps = s;
        while (*ps != 0x0D) {  \(2\)
            CompStrBuf (cpmsg, *ps, GS_TICKS_PER_SRT);
            CRTTimeStr (1);
            ps++;
        }
        ctr++;
    }
}
```


Listing 1.21  TestRxTask()

```c
void TestRxTask (void *data) {
    char *ps;
    char *bytes;
    char s[81];

    data = data;
    for (i; i < 2;)
        bytes = 0;
    do {
        c = ComGetChar (COMM, CB_PCOM_MESSAGE, &err); (1)
        *ps++ = c; (2)
        bytes++; (3)
    } while (c != 0x0a && bytes < 20);
    *ps = 0; (3)
    PC_Display (49, 17, s, DISP_FONT_YELLOW + DISP_FONT_BLUE); (4)
}
```

1.08 Bibliography

JDR Microdevices
1850 South 10th Street
San Jose, CA 95112-4108
(800) 538-5000
(408) 494-1400

PDS-601 link:
Listing 1.22 CFG.C

/*
 * Complete and Ready-to-Use Modules in C
 * Configuration File
 *
 * (C) Copyright 1999, Jean J. Labrosse, Weston, FL
 * All Rights Reserved
 *
 * Filename: CFG.C
 * Programmer: Jean J. Labrosse
 *
 */

#include "includes.h"

/* Prog */
Listing 1.22 (continued)  CFG.C

```c
/*
 * DIGITALIZ I/O PORTS
 */

void KeyInPort (void)
{
    output(HIGH_ROW, 0x00);       /* Initialize R255A: A=OUT, B=IN (255), C=OUT (R09) */
}

/
+

#   SELECT A ROW
+
#
* Description: This function is called to select a row on the keyboard.
* Arguments: row is the row number (0...7) or KEY_ALL_ROW.
* Returns: none
* Notes: The row is selected by writing a LOW.
*/

void KeySelect (unsigned row)
{
    if (row == KEY_ALL_ROW) {
        output(HIGH_MORE, 0x00);           /* Force all rows LOW */
    } else {
        output(HIGH_MORE, -1 < row);       /* Force desired row LOW */
    }
}

#
+
#   READ A COLUMN
+
#
* Description: This function is called to read the column port.
* Arguments: none
* Returns: the complement of the column port that was last key pressed.
*/

dWord keyGetCol (void)
{
    return (!inp(HIGH_PORT_COL));      /* Complement columns (ones indicate key is pressed) */
}

exit(0);
/*Exit*/
```
Listing 1.22 (continued) CFG.C

/ *  
* MULTIPLEXED LED DISPLAY  
* I/O PORTS INITIALIZATION  
*  
* Description: This is called by InitMain() to initialize the output ports used in the LED multiplexing.  
* Arguments: none  
* Returns: none  
* Notes: 74LS273 8-bit latches are used for both the segments and digits outputs.  
*====================================================================*/

#define LED port

void DisplayInit (void)  
  
  {  
    outp(LED_PORT_SEG, 0x00); /* Turn OFF segments */
    outp(LED_PORT_DIG, 0x00); /* Turn OFF Digits */
  }

/ */

/*====================================================================*/

void SegOutSeg (unsigned seg)  
  
  {  
    outp(LED_PORT_SEG, seg);
  }

/*====================================================================*/

void DigitOut (unsigned digit)  
  
  {  
    outp(LED_PORT_DIG, digit);
  }  

/*
*/

/ *  
* MULTIPLEXED LED DISPLAY  
* Digits output  
*  
* Description: This function outputs digit-selector.  
* Arguments: none  
* Returns: none  
* Notes:  
*====================================================================*/

/*
*/
Listing 1.22 (continued)  CFG.C

/*
 * LCD DISPLAY MODULE
 * INITIALIZE DISPLAY DRIVER & I/O PORTS
 * Description: This initializes the I/O ports used by the display driver.
 * Arguments: none
 * Returns: none
 */

#ifndef MODEL_LCD

void DisplayInit (void)
{
    output(LCD_PORT_OUT, 0x82); /* Set to Mode 0: I are output, R are inputs, C are outputs */
}

/*
 * LCD DISPLAY MODULE
 * WRITE DATA TO DISPLAY DEVICE
 * Description: This function sends a single byte to the display device.
 * Arguments: data  - the data to be sent to the display device
 * Returns: none
 * Notes: You will need to adjust the value of LCD_DELAY (125 uS) to produce a delay between
 *        writing of at least 60 uS. The display I used for the test actually required a delay of
 *        80 uS if characters even to appear randomly on the screen, you might need to increase
 *        the value of LCD_DELAY.
 */

void DisplayWrite (unsigned char data)
{
    output(LCD_PORT_DATA, data); /* Write data to display module */
    output(LCD_PORT_DATA, 0x44); /* Set R line HIGH */
    display(); /* Delay about 1 uS */
    output(LCD_PORT_DATA, 0x40); /* Set R line LOW */
    output(LCD_PORT_DATA, 0x00); /* Delay for at least 40 uS */
    display();
}

#endif // MODEL_LCD
Listing 1.22 (continued) CFG.C

/*
 * LCD DISPLAY MODULE
 * 
 * Description: This function read a byte from the display device.
 * 
 * Arguments: none
 */

void readByte (unsigned int addr)
{
    if (addr == DEFP_WIRC_NOR_REG)
        /* Select the command register (Hi low)
        */
        outp(EP_WIRC_NOR, 0x60);
    else
        /* Select the data register (Hi low)
        */
        outp(EP_WIRC_NOR, 0x60);
}

endif

/*
 * MODEM MODULE
 * 
 * DESCRIPTION:
 */

#endif

/*
 * TOWER MODULE
 * 
 */

#endif

/*
 * DSK32
 * 
 */
Listing 1.22 (continued) CFG.C

/**
 * @brief I/O MODULE
 * @brief INITIALIZE PHYSICAL I/O
 *
 * @description This function is called by DECSys() to initialize the physical I/O used by the DE driver.
 * @arguments None.
 * @returns None.
 * @notes The physical I/O is assumed to be an 8255A chip initialized as follows:
 * Port A = OUT (discrete outputs) (Address 0x0000)
 * Port B = IN (discrete inputs) (Address 0x0001)
 * Port C = IN (I/O Data) (Address 0x0002)
 * Control word (Address 0x0003)
 * Refer to the Intel 8255A data sheet.
 */

void DECSysID(void)
{
    outp(0x0000, 0x02); /* Port A = OUT, Port B = IN, Port C = OUT */
}

/**
 * @brief I/O MODULE
 * @brief ECHO PHYSICAL INPUTS
 *
 * @description This function is called to read and map all of the physical inputs used for discrete
 * inputs and map these inputs to their appropriate discrete input data structure.
 * @arguments None.
 * @returns None.
 */

void DEI1D(void)
{
    DECI3 *ph1;
    DEHI i;
    DEIO in;
    DEIO out;

    ph1 = DECI3(0x30);
    in = DEHI(); /* Read all 8 bits to first 8 I/O channels */
    for (i = 0; i < 8; i++) {
        if ((in[i] & 0x80) != 0)
            ph1->out[i] = (ph1->out[i] & 0x00FF) | 0x01; /* Map all 8 bits to first 8 I/O channels */
    }
}
Listing 1.22 (continued) CFG.C

/*
 * Description: This function is called to map all of the discrete output channels to their appropriate physical destinations.
 * Arguments:  None.
 * Returns:    None.
 */

void DIOE (void)
{
    struct DIOE *dioe;
    INMRB; /*
    INMRB; out;
    INMRB; in;
    
    pdo = (MFRb&0x3);
    rsk = (MFRb>>2);
    out = (rsk);
    for (i = 0; i < 8; i++) { /*
        if ((pdo>>i & 0x00) == (rsk)) /*
            out = pdo;
        pdo++; /*
    }
    outp(out); /*
    */
    
    /** Output port image to physical port */
    } #endif
    /**MRB/*
Listing 1.22 (continued) CFG.C

/*

******************************************************************************

* ARM70 I/O MODULE

* INITIALIZE PHYSICAL I/Os

* Description: This function is called by ARMInit() to initialize the
physical I/O used by the ADC
* Arguments: None.
* Returns: None.

*******************************************************************************/

/*

******************************************************************************

* MODELES

* void MODELES(void)

* { /* This is where you will put your initialization code for the ADCs and
DIOs */
*      /* You should also consider initializing the contents of your
I/O register to a known value. */
* }

*******************************************************************************/

/*

******************************************************************************

* ARM70 I/O MODULE

* READ PHYSICAL I/Os

* Description: This function is called to read a physical ADC channel. The
function is assumed to
also control the multipler if more than one analog input is connected to the
ADC.
* Arguments: chc = the ADC logical channel number (0, ADC0, ADC1, -)
* Returns: value: The raw ADC counts from the physical device.

*******************************************************************************/

/*

******************************************************************************

* INT16U ADM(int8_t chc)

* { /* This is where you will put the code to read your ADC(s). */
*      /* ADM() is passed a logical channel number. You will have to
calculate this logical channel */
*      /* number into actual physical port locations by addressing where your
MUX and ADCs are located. */
*      /* ADM() is responsible for: */
*      /* 1) Selecting the proper MUX channel. */
*      /* 2) Waiting for the MUX to stabilize. */
*      /* 3) Starting the ADC. */
*      /* 4) Waiting for the ADC to complete its conversion. */
*      /* 5) Reading the counts from the ADC and. */
*      /* 6) Returning the counts to the calling function. */

* return counts;

* } /*ADM*/
Listing 1.22 (continued)  CFG.C

/*
 ** ANMOC I/O MODULE
 */

/* Description : This function is called to write the "raw" counts to the proper analog output device
 ** (i.e. DAC). It is up to this function to direct the DAC counts to the proper DAC if more
 ** than one DAC is used.
 ** Arguments : ch  is the DAC logical channel number (1...4294967295).
 **             cnts are the DAC counts to write to the DAC.
 ** Returns : None.
 *********************************************************************************/

void ANMc (unsigned long cnts)
{
    ch = ch;
    cnts = cnts;
    /* This is where you will need to provide the code to update your DAC(s). */
    /* ANMc is passed a "LOGICAL" channel number. You will have to convert this logical channel
    ** number into actual physical port locations (or addresses) where your DACs are located. */
    /* ANMc is responsible for writing the counts to the selected DAC based on a logical number. */
    #endif
# define KEY_BUF_SIZE 10 /* Size of the KEYBUF buffer */
# define KEY_BTN_MIN 4 /* The minimum number of rows on the keyboard */
# define KEY_BTN_MAX 6 /* The maximum number of columns on the keyboard */
# define KEY_PORT_MIN 0x0312 /* The port address of the keyboard matrix I/O */
# define KEY_PORT_MAX 0x0313 /* The port address of the keyboard matrix I/O */
# define KEY_PORT_CN 0x0313 /* The port address of the 3.0 ports control word */
# define KEY_BT_MIN 20 /* Number of scans before auto repeat executes again */
# define KEY_BT_MAX 100 /* Number of scans before auto repeat function engages */
# define KEY_SCANCODE_MIN 50 /* Number of scan keys mapped to keyboard scans */
# define KEY_SCANCODE_MAX 50 /* Get priority of keyboard scan task */
# define KEY_SCANCODE_BUF_SIZE 1024 /* Size of keyboard scan TASK stack */
# define KEY_SHIFT1_MSK 0x80 /* The SHFTI key is on bit 7 of the column input port */
# define KEY_SHIFT1_OFFSET 28 /* The scan code offset to add when SHIFT1 is pressed */
# define KEY_SHIFT2_MSK 0x40 /* The SHFT2 key is on bit 6 of the column input port */
# define KEY_SHIFT2_OFFSET 40 /* The scan code offset to add when SHIFT2 is pressed */
# define KEY_SHIFT_MSK 0x00 /* The SHFTI key is on bit 6 of the column input port */
# define KEY_SHIFT_OFFSET 40 /* The scan code offset to add when SHIFT2 is pressed */
#endif /* CNODE */
Listing 1.23 (continued)  CFG.H

******************************************************************************
* MULTIPURPOSE LCD DISPLAY DRIVER CONFIGURATION CONSTANTS
* (Chapter 4)
******************************************************************************
/

ifdef MODULE_LCD
#define DEPS_PORT_SEG 0x3100 /* Port address of SEGMENS output */
#define DEPS_PORT_DSEG 0x3101 /* Port address of DSEGTS output */
#define DEPS_DSEG 8 /* Total number of digits (including status indicators) */
#define DEPS_DSEG 8 /* Total number of seven-segment digits */
endif
/

******************************************************************************
* LCD DISPLAY MODULE DRIVER CONFIGURATION CONSTANTS
* (Chapter 5)
******************************************************************************
/

ifdef MODULE_LCD
#define LCD_DELAY 100 /* Number of iterations to delay for 40 us (software loop) */
#define LCD_PORT_DSEG 0x3100 /* Port address of the G7A part of the LCD module */
#define LCD_PORT_DSEG 0x3103 /* Address of the Control Word (RCC55) to control RS & E */
endif
/

""
Listing 1.23 (continued) CFG.H

/*
 * CLOCK/CHIPTERM MODULE CONFIGURATION CONSTANTS
 */

#include <CFG.h>

#define CLC_CLK_PRI 40  /* This defines the priority of Clk[0:9] */
#define Clc_CLK_TICKS 960  /* # of clk ticks to obtain & second */
#define Clc_CLK_TICK_SIZE 512  /* # of clk ticks in bytes for Clk[0:9] */

#define Clc_CLK_CYCLE 1  /* Enable DATE (when 1) */
#define Clc_CLK_CYCLE 1  /* Enable TIME-STAMPS (when 1) */
#define Clc_CLK_CYCLE 1  /* Use Clc_CLK cycle instead of pend on sect. */

#include <TMR.h>

#define TMR_CLK_PRI 40  /* This defines the priority of Tmr[0:9] */
#define TMR_CLK_TICKS 1024  /* # of clk ticks to obtain & second */
#define TMR_CLK_TICK_SIZE 512  /* # of clk ticks in bytes for Tmr[0:9] */
#define TMR_CLK_CYCLE 0  /* Use TMR_CLK cycle instead of pend on sect. */

*/
List 1.23 (continued)  CFG.H

/
**************************************************************************/
 Chap 8
**************************************************************************/
/

#define DIO_DARK_PRO 35
#define DIO_DARK_BK 35
#define DIO_DARK_DARK 517
#define DIO_MAX_DK 8 "Max. number of Discrete Input Channels (1...255)"
#define DIO_MAX_DD 8 "Max. number of Discrete Output Channels (1...255)"
#define DC_READ_EN 1 "Enable code generation to support edge trig. (when 1)"
#define DC_READ_EN 1 "Enable code generation to support blink code (>000 1)"

#endif

/
**************************************************************************/
 Chap 12
**************************************************************************/
/

#define AIO_DARK_PRO 10
#define AIO_DARK_BK 100 "Execute every 100 mS"
#define AIO_MAX_AI 8 "Max. number of Analog Input Channels (1...250)"
#define AIO_MAX_AD 8 "Max. number of Analog Output Channels (1...250)"

#endif

"END"/
Listing 1.23 (continued)  CFG.H

`/*
 * AUTOMOTIVE SERIAL COMMUNICATIONS MODE CONFIGURATION C:
 * Chapter III
 */

#define CONN_BASE 0x03F0 /* base address of PC's COM1 */
#define COML_BASE 0x03F0 /* base address of PC's COM2 */
#define COM1_UART_BUF 2 /* maximum number of characters in rx buffer of ... */
/* ... MSDOS UART: 2 for 14400, 16 for 16550. */
#endif

#define MODULE_COM2_RCV 1
#define COM2_BASE 2
#define COM2_UART_BUF 64 /* number of characters in rx ring buffer */
#define COM2_RX_BUF_SIZE 64 /* number of characters in rx ring buffer */
#endif

#define MODULE_COM3_RCV 1
#define COM3_BASE 7
#define COM3_UART_BUF 64 /* number of characters in rx ring buffer */
#define COM3_RX_BUF_SIZE 64 /* number of characters in rx ring buffer */
#endif`
Listing 1.24  INCLUDES.H

/*
   ********************************************
   * Embedded Systems Building Blocks      *
   * Complete and Ready-to-Use Modules in C *
   *                                           *
   * Header Include File                      *
   *                                           *
   * Copyright 1999, Dean J. Lurie, Warren, MI *
   * All Rights Reserved                      *
   * Filename : INCLUDES.H                     *
   * Programmer : Dean J. Lurie               *
   *                                           *
   ********************************************/ /*

/*
   CONSTANTS
   */

#define MODULE_APP_NM 1 /* Module Enabled (1) or Disabled (0) */
#define MODULE_LED 0 /* LED Character module */
#define MODULE_LCD 1 /* LED Character module */
#define MODULE_CLK 1 /* Clock/Calendar module */
#define MODULE_TIMER 1 /* Timer/Moniter module */
#define MODULE_IO 1 /* Discrete I/O module */
#define MODULE_ADC 1 /* Analog I/O module */
#define MODULE_COMM_1 1 /* Asynchronous Serial Communications module */
#define MODULE_COMM_2 1 /* Bidirectional Serial I/O */
#define MODULE_COMM_3 1 /* Real-Time Kernel Insulated serial I/O */
#define MODULE_DATA 1 /* Slapped time measurement module */
#define CPU_L 0 /* Indicates that application specific code is found in CPU.L */
#define CPU_H 1 /* Indicates that configuration defines is found in CPU.H */

/*
   
   */

#define FALSE 0
#define TRUE 1

/*SINCE*/
Listing 1.24 (continued) INCLUDES.H

/
-------------------------------------------------------------------
* Standard libraries (C89)
-------------------------------------------------------------------
/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#include <winreg.h>

/
-------------------------------------------------------------------
* UC/OS Header Files
-------------------------------------------------------------------
/
#include "software/ucos-ii.lib-symbian310a.spu.h"
#include "software/ucos-ii.lib-symbian844a.spu.h"
#include "software/ucos-ii.lib-base.spu.h"

/
*/

"GROUP"
Listing 1.24 (continued) INCLUDES.H

/** 
 * Building Blocks Header File
 */
#
include "software/block/sample1/source/cott.h"
#include "software/block/sample2/source/cott.h"

if MODULE_KEY
#include "software/block/key/sample1/source/cott.h"
#include "software/block/key/sample2/source/cott.h"
endif

if MODULE_LCD
#include "software/block_lcd/sample1/source/cott.h"
#include "software/block_lcd/sample2/source/cott.h"
endif

if MODULE_IAP
#include "software/block_iap/sample1/source/cott.h"
#include "software/block_iap/sample2/source/cott.h"
endif

if MODULE_DSF
#include "software/block_dsfsample1/source/cott.h"
#include "software/block_dsfsample2/source/cott.h"
endif

if MODULE_APC
#include "software/block_apcsample1/source/cott.h"
#include "software/block_apcsample2/source/cott.h"
endif

if MODULE_COMM_PC
#include "software/block_cottsample1/source/cott_pc.h"
#include "software/block_cottsample2/source/cott_pc.h"
endif

if MODULE_COMM_SPI
#include "software/block_cottsample1/source/cott_spi.h"
#include "software/block_cottsample2/source/cott_spi.h"
endif

if MODULE_COMM/gpio
#include "software/block_cottsample1/source/cott_gpio.h"
#include "software/block_cottsample2/source/cott_gpio.h"
endif

/*
Listing 1.25  \texttt{MAKETEST.BAT}

```
ECHO OFF
CLS
ECHO **************************************************************************
ECHO Embedded Systems Building Blocks
ECHO *
ECHO * (C) Copyright 1999, Jesse C. Jabrabe, Weston, FL
ECHO * All Rights Reserved
ECHO *
ECHO *
ECHO * Filename : \texttt{MAKETEST.BAT}
ECHO * Description : batch file to create the application.
ECHO * Output : \texttt{TMP\_EXE} will contain the EXE executable
ECHO * Usage : \texttt{MAKETEST}
ECHO * Note(s) : 1) This file assumes that we use a MASM utility.
ECHO **************************************************************************
ECHO *
ECHO *
ECHO OF
MD ..\MAKE
MD ..\OBJ
MD ..\LINK
CD ..\MAKE
COPY ..\TEST\TEST\\texttt{NEX}\TEST\\texttt{NEX}
COPY ..\TEST\TEST\\texttt{NN}\TEST\\texttt{NN}
IF NOT EXIST \texttt{TEST.BAT} \COPY \texttt{TEST.BAT} \texttt{TEST.BAT}
CALL \texttt{TEST.BAT}
CD ..\TEST
```
Listing 1.26 OS_CFG.H

/*
 * OS_CFG.H
 */

/*
 * Configuration for Intel 80486 (Large)
 */

/*
 * Defines:
 */

#define OS_MAX_EVENTS 5 /* Max. number of event control blocks in your application */
#define OS_MAX_HBAR_SET 5 /* Max. number of memory partitions */
#define OS_MAX_QCT 5 /* Max. number of queue control blocks in your application */
#define OS_MAX_TASKS 10 /* Max. number of tasks in your application */
#define OS_LOWEST_PRIO 63 /* Defines the lowest priority that can be assigned */

#define OS_MAX.toFloat 92 /* Define the stack size (# of 16-bit words entries) */
#define OS_MAX_STACK_SIZE 512 /* Define the stack size (# of 16-bit words entries) */
#define OS_MAX_STATS_SIZE 512 /* Define the statistics stack size (# of 16-bit words entries) */
#define OS_MAX_PORTS 1 /* Define the maximum number of processor port files */
#define OS_MAX_TIME_OUT 1 /* Include code for MAX_TIMESTAMP */
#define OS_MAX_MEMORY 1 /* Include code for MEMORY_MANAGER (fixed sized memory block) */
#define OS_MAX_TIME_OUT 1 /* Include code for INCURSES */
#define OS_MAX_TIME_OUT 1 /* Include code for DECURSES */
#define OS_MAX_TASK_SET 0 /* Include code for OFFThreadStart() */
#define OS_MAX_TIME_OUT 1 /* Include code for OFFThreadDestroy() */
#define OS_MAX_TIME_OUT 1 /* Include code for OFFThreadStart() */
#define OS_MAX_TIME_OUT 1 /* Include code for OFFThreadDestroy() */
#define OS_MAX_TIME_OUT 1 /* Set the number of ticks in one second */
Listing 1.27  TEST.C

/************************************************************************
 *          Embedded Systems Building Blocks
 *          Complete and Ready-to-Use Modules in C
 *          
 *          (c) Copyright 1995, Jean J. Lebrasse, Westcon, PL.
 *          All Rights Reserved.
 *
 * Filename : TEST.C
 * Programmer : Jean J. Lebrasse
 ************************************************************************/

#include "includes.h"

/************************************************************************
 * CONFIGURATION
 ************************************************************************/

#define TASK_STACK_SIZE 512 /* Size of each task's stack (# of 16-bit words) */
#define TASK_MAX_PRIO 10
#define TASK_MIN_PRIO 20
#define TASK_MAX_PRIO 30

/************************************************************************
 * VARIABLES
 ************************************************************************/

OIL_OPN TestStartTask(TASK_STACK_SIZE);
OIL_OPM TestStartTask(TASK_STACK_SIZE);
OIL_OPM TestStartTask(TASK_STACK_SIZE);

/************************************************************************
 * FUNCTION PROTOTYPES
 ************************************************************************/

void TestStartTask(void *data);
void TestTask(void *data);
void TestStartTask(void *data);

static void TestStartTask(void);
static void TestStartTask(void *data);
static void TestStartTask(void *data);

/* main */
Listing 1.27 (continued)  TEST.C

void main (void)
{
   char *str = "This is a string.
   char *str2 = 0;
}

Chapter 1: Sample Code — 45
Listing 1.27 (continued)  TEST.C

```c
/*
 * FUNCTION: TASK
 *
 * void TaskGen(void) {
 *    char *task
 *    ...
 * }
 *
 * void TestThread(void *data) {
 *    char *thread
 *    ...
 * }
 */
```

```c
pdata = pdata; /* Prevent compiler warning */

PC_Dispatch(0, 0, "EMBEDDED SYSTEMS BUILDING BLOCKS ",
          DESP_FORE_WRITE + DESP_FORE_READ + DESP_FORE_BLACK);

PC_Dispatch(1, 0, "Complete and Ready-to-Use Module in C",
          DESP_FORE_WRITE + DESP_FORE_READ + DESP_FORE_BLACK);

PC_Dispatch(2, 1, "J. W. Labrosse",
          DESP_FORE_WRITE + DESP_FORE_READ + DESP_FORE_BLACK);

PC_Dispatch(3, 0, "SAMPLE COMP.",
          DESP_FORE_WRITE + DESP_FORE_READ + DESP_FORE_BLACK);

DELAY(3000); /* Install aTICK-1’s clock tick ISR */

PC_SetUpTime(EVENT_TIME_FUN(WK)); /* Begin program tick rate */

void TestThread(void *) {
    char *thread
    ...
}
```

```c
for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        for (k = 0; k < 5; k++) {
            ...
        }
    }
```
Listing 1.27 (continued) TEST.C

RC(timeGetTime());  /* Get and display date and time */
RC_SetPrintf(24, 0, DISP_MONDAY | DISP_MONDAY | DISP_MONDAY);  
if (RC_GetKey(Key) == TRAP)  /* See if key has been pressed */
    if (Key == Esc)  /* Yes, see if it's the ESCAPE key */
        RC_XClearScreen();  /* Return to DOS */
    }
    CTime(0, 0, 0);  /* Wait one second */
} /*MAIN*/
Listing 1.27 (continued) TEST.C

/*
   This task displays numbers randomly.
   
   Task uses random numbers and
   data.
*/

void TestTask(void *data)
{
  char *data;
  0x0140 time;

  data = data;
  PC_display(0, 5, "data ", *DIGI_FONT_MEDIUM);
  PC_display(0, 7, "data ", *DIGI_FONT_MEDIUM);
  PC_display(0, 9, "data ", *DIGI_FONT_MEDIUM);
  PC_display(0, 8, "data ", *DIGI_FONT_MEDIUM);
  PC_display(0, 0, "data ", *DIGI_FONT_MEDIUM);
}

/* Prevent compiler warning */

TestInitModules();

/* Initialize all building blocks used */

CIMEDate(time(13, 31, 1999, 23, 57, 55));

/* Set the clock/calendar */

/* Execute when Timer B timer out */

/* Execute when Timer A timer out */

/* Set timer B to 1 mSec. 3 sec. 9/10 sec. */

/* Set timer A to 2 minutes */

/* Initialize discrete outputs #3 and #1 */

DOCCaille(0, DD_OUTPUT_3, 0, 18);

DOCCaille(0, DD_OUTPUT_4, 0, 35);

DOCCaille(0, DD_OUTPUT_5, 0, 65);

DOCCaille(0, DD_OUTPUT_6, 0, 65);

DOCCaille(0, DD_OUTPUT_7, 0, 65);
Listing 1.27 (continued) TEST.C

for (i = 0; i < 2; i++)
{
    PC_ElapsedStart();
    CMvformatDate(2, s);
    /* Get formatted date from clock/calendar */
    PC_ElapsedStop();
    PC_Elapsed(10, 6, 8, DEEP_PDIR_MIDDLE);
    sprintf(s, "CMvformatDate() takes %d ms", time);
    PC_ElapsedStart();
    CMvformatTime(1, s);
    /* Get formatted time from clock/calendar */
    PC_ElapsedStop();
    PC_Elapsed(10, 7, 6, DEEP_PDIR_MIDDLE);
    sprintf(s, "CMvformatTime() takes %d ms", time);
    PC_ElapsedStart();
    PC_Elapsed(8, 6, 6, DEEP_PDIR_MIDDLE);
    Perimeter(s, a);
    /* Get formatted remaining time for 2s-KU */
    PC_ElapsedStart();
    PC_Elapsed(8, 6, 8, DEEP_PDIR_MIDDLE);
    Perimeter(s, a);
    /* Get formatted remaining time 2s-KU */
    PC_ElapsedStart();
    PC_Elapsed(9, 8, 6, DEEP_PDIR_MIDDLE);
    PC_ElapsedStart();
    PC_Elapsed(10, 0, 0, DEEP_PDIR_MIDDLE);
    /* Display state of Byrne's outputs */
    PC_ElapsedStart();
    PC_Elapsed(0, 10, 0, DEEP_PDIR_MIDDLE);
    /* Display state of Byrne's outputs */
    CMvtime(0x000, 0, 0, 100);
}

/*Done*/
Listing 1.27 (continued) TEST.C

/*
 * 
 * 
 * 
 * 
 */

void testTask(void *data)
{
    int x;
    int y;
    int z;

    data = data;
    for (y = 0; y < 10; y++) {
        x = rand() % 10; /* Find x position where task number will appear */
        y = rand() % 10; /* Find y position where task number will appear */
        z = rand() % 10; /* Find random number from 0 to 9 */
        #ifdef DEBUG
        printf("%d, %d, %d\n", x, y, z); /* DEBUG 통해서 출력 */
        #endif
    }
}

/*INVP*/
Listing 1.27 (continued) TEST.C

/*
******************************************************************************
* EMBEDDED SYSTEMS BUILDING BLOCKS
* Module Initialization
*******************************************************************************/

static void TestInitModules(void)
{
#if MODULE_ELAPSED
   PC.ElapsedInit();
#endif
#if MODULE_KEYBOARD
   KeyScanInit();
#endif
#if MODULE_LCD
   DispInit(4, 20);
#endif
#if MODULE_RTC
   CIRTClock();
#endif
#if MODULE_CLOCK
   TimeInit();
#endif
#if MODULE_PWM
   PWMInit();
#endif
#if MODULE_A2D
   A2DInit();
#endif
#if MODULE_COMS_PC
   ConCInit(CONSOLE, 5600, B, COM2_SPEED_9600, 1);
#endif
#if MODULE_COMS_RS232
   ConCInit();
#endif
#if MODULE_UART
   uartInit();
#endif
}
} /*END*/
Listing 1.27 (continued)  TEST.C

/* *******************************************************
 * Function executed when Times Time Out.
 * *******************************************************
 */

static void TestCreate (void *arg) {
  arg = arg;
  PC_Finger(12, 8, "Timer #6 Tired Out!", DISP_FND_TIME);
}

static void TestCreate2 (void *arg) {
  arg = arg;
  PC_Finger(12, 9, "Timer #7 Tired Out!", DISP_FND_TIME);
}
Listing 1.28 TEST.LNK

```
pg 1 of 15 P:\LIB\TEST.LNK

- VersAnt_ORD.OBJ
- VersAnt_ORD.BLD
- VersAnt_ORD.GEN
- VersAnt_ORD(PC).OBJ
- VersAnt_ORD(PC).GEN
- VersAnt_ORD(PC).BLD
- VersAnt_ORD.Int.BLD
- VersAnt_Def.OBJ
- VersAnt_Def.BLD
- VersAnt_Def.GEN
- VersAnt_Def_Bld_OPTIONS.OBJ
- VersAnt_Def_Bld_OPTIONS.BLD
- VersAnt_Def_Bld_OPTIONS.GEN
- VersAnt_Def_Bld_OPTIONS.GEN
- VersAnt_Def_Bld_OPTIONS.BLD
- VersAnt_Def_Bld_OPTIONS.OBJ
- VersAnt_Def_Bld_Options...VersAnt_Def...VersAnt_Def_Def.OBJ
- VersAnt_Def_Def.OBJ
- VersAnt_Def_Def.BLD
- VersAnt_Def_Def.GEN
- VersAnt_Def_Def.GEN
- VersAnt_Def_Def.BLD
- VersAnt_Def_Def.OBJ

spec.LBL
```
Listing 1.29 TEST.MAK

# Embedded Systems Building Blocks
# (c) Copyright 1999, Jason J. Lebrun, Weston, FL
# All Rights Reserved
#
#
# FileName : TEST.MAK

# "BEGIN"
#
##
##
CGE= : CC123456.DECC
ASR= : VCC123456.DECC
LDL= : MVE123456.DECC

# "END"

#.

# TARGET
#.
SOURCE = \VCHANGE
TEST = \VVALUE
MAKE = \VMAKE
CC = \VCC
LD = \VLD

#.
ASR = \VCHANGE
CGE = \VCHANGE
LDDL = \VCHANGE

#.
LDL = \VCHANGE
ASR = \VCHANGE
CGE = \VCHANGE

#.
LDL = \VCHANGE
ASR = \VCHANGE
CGE = \VCHANGE

#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
#.
Listing 1.29 (continued) TEST_MAX

#pragma pack

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define PI 3.14159265358979323846

int main()
{
    int i = 0;
    double sum = 0.0;
    for (i = 0; i < 1000000; i++)
    {
        sum += (i % 6) ? PI : i * 0.5;
    }
    printf("%f\n", sum);
    return 0;
}
Listing 1.29 (continued) TEST.MAK

******************************************************************************
# CREATION OF .OBJ FILE
******************************************************************************

$ECHO -E TEST.EXE
$ECHO -E VAR.OBJ
$ECHO -E CPRO.LIB
$ECHO -E COB.OBJ
$ECHO -E COB.COB
$ECHO -E COM Chỉ.COB
$ECHO -E CODE.COB
$ECHO -E CODE.COB
$ECHO -E CODE.COB
$ECHO -E CODE.COB
$ECHO -E CODE.COB
$ECHO -E CODE.COB

LIBRARIES
LIB
COPY $ECHO -E TEST.LIB
DEL $ECHO -E TEST.MAK
DEL $ECHO -E TEST.MAK
DEL $ECHO -E TEST.MAK
COPY LOCAL.COB -TEST.MAK  /V
COPY LOCAL.COB -TEST.MAK  /V
COPY LOCAL.COB -TEST.MAK  /V
DEL TEST.MAK

******************************************************************************
# CREATION OF .OCC (Object) FILES
******************************************************************************

$ECHO -E OCC.OBJ
$ECHO -E OCC.OBJ
$ECHO -E OCC.OBJ
COPY $ECHO -E OCC.OBJ
DEL $ECHO -E OCC.OBJ
COPY $ECHO -E OCC.OBJ
DEL $ECHO -E OCC.OBJ
COPY $ECHO -E OCC.OBJ
DEL $ECHO -E OCC.OBJ

******************************************************************************
# CREATION OF .C Source FILES
******************************************************************************

$ECHO -E C.C
COPY $ECHO -E C.C
DEL $ECHO -E C.C
COPY $ECHO -E C.C
DEL $ECHO -E C.C
COPY $ECHO -E C.C
DEL $ECHO -E C.C
Listing 1.29 (continued) TEST.MAK

$COMMD.COMMPC.ORG: $COMMD.COMMPC.C \n  INCLUDE: H
  COPY $COMMD.COMMPC.C
  OBJ $COMMD.COMMPC.OBJ
  $($(CC) $$(CFLAGS) $COMMD.COMMPC.OBJ)

$COMMD.COMMPCL.ORG: $COMMD.COMMPCL.ADM
  COPY $COMMD.COMMPCL.ADM
  OBJ $COMMD.COMMPCL.OBJ
  $$(ASM) $COMMPCL.LOAD\$(CC) $COMMD.COMMPCL.ADM $COMMD.COMMPCL.OBJ

$COMMD.COMMPCOS.ORG: $COMMD.COMMPCOS.C \n  INCLUDE: H
  COPY $COMMD.COMMPCOS.C
  OBJ $COMMD.COMMPCOS.OBJ
  $$(CC) $$(CFLAGS) $COMMD.COMMPCOS.OBJ

$$(CC) $$(CFLAGS) $COMMD.COMMPCOS.OBJ

$COMMD.COMPS.ORG: $COMMD.COMPS.C \n  INCLUDE: H
  COPY $COMMD.COMPS.C
  OBJ $COMMD.COMPS.OBJ
  $$(CC) $$(CFLAGS) $COMMD.COMPS.OBJ

$COMMD.COMPS.DAT: $COMMD.COMPS.C \n  INCLUDE: H
  COPY $COMMD.COMPS.C
  OBJ $COMMD.COMPS.OBJ
  $$(CC) $$(CFLAGS) $COMMD.COMPS.OBJ

$$(CC) $$(CFLAGS) $COMMD.COMPS.OBJ

$COMMD.LCD.ORG: $COMMD.LCD.C \n  INCLUDE: H
  COPY $COMMD.LCD.C
  OBJ $COMMD.LCD.OBJ
  $$(CC) $$(CFLAGS) $COMMD.LCD.OBJ

$COMMD.LCD.DAT: $COMMD.LCD.C \n  INCLUDE: H
  COPY $COMMD.LCD.C
  OBJ $COMMD.LCD.OBJ
  $$(CC) $$(CFLAGS) $COMMD.LCD.OBJ

$COMMD.LCD2.ORG: $COMMD.LCD2.C \n  INCLUDE: H
  COPY $COMMD.LCD2.C
  OBJ $COMMD.LCD2.OBJ
  $$(CC) $$(CFLAGS) $COMMD.LCD2.OBJ

$COMMD.LCD2.DAT: $COMMD.LCD2.C \n  INCLUDE: H
  COPY $COMMD.LCD2.C
  OBJ $COMMD.LCD2.OBJ
  $$(CC) $$(CFLAGS) $COMMD.LCD2.OBJ

$COMMD.LCD2.DAT: $COMMD.LCD2.ADM
  COPY $COMMD.LCD2.ADM
  OBJ $COMMD.LCD2.ADM
  $$(AS) $COMMD.LCD2.ADM
  $$(CC) $COMMD.LCD2.ADM $COMMD.LCD2.DAT
Listing 1.29 (continued)  TEST.MAK

$(SDK)/VOL_CPU_A.ARM \ 
  INCLUDES
  COPY $(SDK)/VOL_CPU_A.ARM \ OR_CPU_A.ARM
  DEL $(SDK)/VOL_CPU_A.OBY
  $(ASM) $(ASL)$(ASL) \ $(SDK)/VOL_CPU_A2.COR $(SDK)/VOL_CPU_A.MAK

$(SDK)/VOL_CPU_C.C \ 
  INCLUDES
  COPY $(SDK)/VOL_CPU_C.C \ OR_CPU_C.C
  DEL $(SDK)/VOL_CPU_C.OBY
  $(CC) $(C)PLANS \ OR_CPU_C.C

$(SDK)/VOL.C \ 
  INCLUDES
  COPY $(SDK)/VOL.C \ VOL.C
  DEL $(SDK)/VOL.CB2
  $(CC) $(C)PLANS \ VOL.C

$(SDK)/VOL.FS.MAK \ 
  INCLUDES
  COPY $(SDK)/VOL.FS.C \ VOL.FS.C
  DEL $(SDK)/VOL.FSB2
  $(CC) $(C)PLANS \ VOL.FS.C

$(SDK)/VOL.FC.MAK \ 
  INCLUDES
  COPY $(SDK)/VOL.FC.C \ VOL.FC.C
  DEL $(SDK)/VOL.FCB2
  $(CC) $(C)PLANS \ VOL.FC.C

$(SDK)/VOL.FC.MAK \ 
  INCLUDES
  COPY $(SDK)/VOL.FC.C \ VOL.FC.C
  DEL $(SDK)/VOL.FCB2
  $(CC) $(C)PLANS \ VOL.FC.C

$(SDK)/VOL.FC.MAK \ 
  INCLUDES
  COPY $(SDK)/VOL.FC.C \ VOL.FC.C
  DEL $(SDK)/VOL.FCB2
  $(CC) $(C)PLANS \ VOL.FC.C

*(Visited)*
Chapter 2

Real-Time Systems Concepts

Real-time systems are characterized by the severe consequences that result if logical as well as timing correctness properties of the system are not met. There are two types of real-time systems: SOFT and HARD. In a SOFT real-time system, tasks are performed by the system as fast as possible, but the tasks don’t have to finish by specific times. In HARD real-time systems, tasks have to be performed not only correctly but on time. Most real-time systems have a combination of SOFT and HARD requirements. Real-time applications cover a wide range, but most real-time systems are embedded. This means that the computer is built into a system and is not seen by the user as being a computer. The following list shows a few examples of embedded systems.

<table>
<thead>
<tr>
<th>Process control</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food processing</td>
<td>Switches</td>
</tr>
<tr>
<td>Chemical plants</td>
<td>Robots</td>
</tr>
<tr>
<td>Automotive</td>
<td>Aerospace</td>
</tr>
<tr>
<td>Engine controls</td>
<td>Flight management systems</td>
</tr>
<tr>
<td>Anti-lock braking systems</td>
<td>Weapons systems</td>
</tr>
<tr>
<td>Office automation</td>
<td>Jet engine controls</td>
</tr>
<tr>
<td>FAX machines</td>
<td>Domestic</td>
</tr>
<tr>
<td>Copiers</td>
<td>Microwave ovens</td>
</tr>
<tr>
<td>Computer peripherals</td>
<td>Dishwashers</td>
</tr>
<tr>
<td>Printers</td>
<td>Washing machines</td>
</tr>
<tr>
<td>Terminals</td>
<td>Thermostats</td>
</tr>
<tr>
<td>Scanners</td>
<td></td>
</tr>
<tr>
<td>Modems</td>
<td></td>
</tr>
</tbody>
</table>

Real-time software applications are typically more difficult to design than non-real-time applications.

This chapter describes real-time concepts.
2.00 Foreground/Background Systems

Small systems of low complexity are generally designed as shown in Figure 2.1. These systems are called foreground/background or super-loops. An application consists of an infinite loop that calls modules (i.e., functions) to perform the desired operations (background). Interrupt Service Routines (ISRs) handle asynchronous events (foreground). Foreground is also called interrupt level; background is called task level. Critical operations must be performed by the ISRs to ensure that they are dealt with in a timely fashion. Because of this, ISRs have a tendency to take longer than they should. Also, information for a background module made available by an ISR is not processed until the background routine gets its turn to execute. This is called the task level response. The worst case task-level response time depends on how long the background loop takes to execute. Because the execution time of typical code is not constant, the time for successive passes through a portion of the loop is nondeterministic. Furthermore, if a code change is made, the timing of the loop is affected.

Most high-volume microcontroller-based applications (e.g., microwave ovens, telephones, toys, and so on) are designed as foreground/background systems. Also, in microcontroller-based applications, it may be better (from a power consumption point of view) to halt the processor and perform all of the processing in ISRs.
2.01 Critical Section of Code

A critical section of code, also called a critical region, is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted. To ensure this, interrupts are typically disabled before the critical code is executed and enabled when the critical code is finished (see also section 2.03, Shared Resource).

2.02 Resource

A resource is any entity used by a task. A resource can thus be an I/O device, such as a printer, a keyboard, or a display, or a variable, a structure, or an array.

2.03 Shared Resource

A shared resource is a resource that can be used by more than one task. Each task should gain exclusive access to the shared resource to prevent data corruption. This is called mutual exclusion, and techniques to ensure mutual exclusion are discussed in section 2.18, Mutual Exclusion.

2.04 Multitasking

Multitasking is the process of scheduling and switching the CPU (Central Processing Unit) between several tasks; a single CPU switches its attention between several sequential tasks. Multitasking is like foreground/background with multiple backgrounds. Multitasking maximizes the utilization of the CPU and also provides for modular construction of applications. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time applications. Application programs are typically easier to design and maintain if multitasking is used.

2.05 Task

A task, also called a thread, is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area (as shown in Figure 2.2).

Each task typically is an infinite loop that can be in any one of five states: DORMANT, READY, RUNNING, WAITING (for an event), or ISR (interrupted) (Figure 2.3). The DORMANT state corresponds to a task that resides in memory but has not been made available to the multitasking kernel. A task is READY when it can execute but its priority is less than the currently running task. A task is RUNNING when it has control of the CPU. A task is WAITING when it requires the occurrence of an event (waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur, time to expire, etc.). Finally, a task is in the ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt. Figure 2.3 also shows the functions provided by μC/OS-II to make a task move from one state to another.
Figure 2.2  Multiple tasks.
2.06 Context Switch (or Task Switch)

When a multitasking kernel decides to run a different task, it simply saves the current task's context (CPU registers) in the current task's context storage area — its stack (Figure 2.2). Once this operation is performed, the new task's context is restored from its storage area then resumes execution of the new task's code. This process is called a context switch or a task switch. Context switching with overhead to the application. The more registers a CPU has, the higher the overhead. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU. Performance of a real-time kernel should not be judged by how many context switches the kernel is capable of doing per second.

2.07 Kernel

The kernel is the part of a multitasking system responsible for the management of tasks (i.e., managing the CPU’s time) and communication between tasks. The fundamental service provided by the kernel is context switching. The use of a real-time kernel generally simplifies the design of systems by allowing the application to be divided into multiple tasks managed by the kernel. A kernel adds overhead to your system because it requires extra ROM (code space) and additional RAM for the kernel data structures. But most importantly, each task requires its own stack space, which has a tendency to eat up RAM quite quickly. A kernel will always consume CPU time (typically between 2 and 5 percent).

Single-chip microcontrollers are generally not able to run a real-time kernel because they have very little RAM. A kernel allows you to make better use of your CPU by providing you with indispensable...
services such as semaphore management, mailboxes, queues, time delays, etc. Once you design a system using a real-time kernel, you will not want to go back to a foreground/background system.

2.08 Scheduler
The scheduler, also called the dispatcher, is the part of the kernel responsible for determining which task will run next. Most real-time kernels are priority based. Each task is assigned a priority based on its importance. The priority for each task is application specific. In a priority-based kernel, control of the CPU is always given to the highest priority task ready to run. When the highest priority task gets the CPU, however, is determined by the type of kernel used. There are two types of priority-based kernels: non-preemptive and preemptive.

2.09 Non-Preemptive Kernel
Non-preemptive kernels require that each task does something to explicitly give up control of the CPU. To maintain the illusion of concurrency, this process must be done frequently. Non-preemptive scheduling also called cooperative multitasking: tasks cooperate with each other to share the CPU. Asynchronous events are still handled by ISRs. An ISR can make a higher priority task ready to run, but the ISR always returns to the interrupted task. The new higher priority task will gain control of the CPU only when the current task gives up the CPU.

One of the advantages of a non-preemptive kernel is that interrupt latency is typically low (see the later discussion on interrupts). At the task level, non-preemptive kernels can also use non-reentrant functions (discussed later). Non-reentrant functions can be used by each task without fear of corruption by another task. This is because each task can run to completion before it relinquishes the CPU. However, non-reentrant functions should not be allowed to give up control of the CPU.

Task-level response using a non-preemptive kernel can be much lower than with foreground/background systems because task-level response is now given by the time of the longest task.

Another advantage of non-preemptive kernels is the lesser need to guard shared data through the use of semaphores. Each task owns the CPU, and you don’t have to fear that a task will be preempted. This is not an absolute rule, and in some instances, semaphores should still be used. Shared I/O devices may still require the use of mutual exclusion semaphores; for example, a task might still need exclusive access to a printer.

The execution profile of a non-preemptive kernel is shown in Figure 2.4. A task is executing [F2.4(1)] but gets interrupted. If interrupts are enabled, the CPU vector jumps (jumps) to the ISR [I2.4(2)]. The ISR handles the event [F2.4(3)] and makes a higher priority task ready to run. Upon completion of the ISR, a Return From Interrupt instruction is executed, and the CPU returns to the interrupted task [F2.4(4)]. The task code resumes at the instruction following the interrupted instruction [F2.4(5)]. When the task code completes, it calls a service provided by the kernel to relinquish the CPU to another task [F2.4(6)]. The new higher priority task then executes to handle the event signaled by the ISR [F2.4(7)].
The most important drawback of a non-preemptive kernel is responsiveness. A higher priority task that has been made ready to run may have to wait a long time to run because the current task must give up the CPU when it is ready to do so. As with background execution in foreground/background systems, task-level response time in a non-preemptive kernel is non-deterministic; you never really know when the highest priority task will get control of the CPU. It is up to your application to relinquish control of the CPU.

To summarize, a non-preemptive kernel allows each task to run until it voluntarily gives up control of the CPU. An interrupt preempts a task. Upon completion of the ISR, the ISR returns to the interrupted task. Task-level response is much better than with a foreground/background system but is still non-deterministic. Very few commercial kernels are non-preemptive.

2.10 Preemptive Kernel

A preemptive kernel is used when system responsiveness is important. Because of this, μC/OS-II and most commercial real-time kernels are preemptive. The highest priority task ready to run is always given control of the CPU. When a task makes a higher priority task ready to run, the current task is preempted (suspended) and the higher priority task is immediately given control of the CPU. If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is resumed and the new higher priority task is resumed. This is illustrated in Figure 2.5.
With a preemptive kernel, execution of the highest priority task is deterministic; you can determine when it will get control of the CPU. Task-level response time is thus minimized by using a preemptive kernel.

Application code using a preemptive kernel should not use non-reentrant functions, unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low- and a high-priority task can use a common function. Corruption of data may occur if the higher priority task preempts a lower priority task that is using the function.

To summarize, a preemptive kernel always executes the highest priority task that is ready to run. An interrupt preempts a task. Upon completion of an ISR, the kernel resumes execution to the highest priority task ready to run (not the interrupted task). Task-level response is optimum and deterministic. μC/OS-II is a preemptive kernel.

2.11 Reentrancy

A reentrant function can be used by more than one task without fear of data corruption. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables (i.e., CPU registers or variables on the stack) or protect data when global variables are used. An example of a reentrant function is shown in Listing 2.1.
Listing 2.1  Reentrant function.

```c
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++)
    {
        ;
    }
    *dest = END;
}
```

Because copies of the arguments to `strcpy()` are placed on the task's stack, `strcpy()` can be invoked by multiple tasks without fear that the tasks will corrupt each other’s pointers.

An example of a non-reentrant function is shown in Listing 2.2. `swap()` is a simple function that swaps the contents of its two arguments. For the sake of discussion, I assume that you are using a preemptive kernel, that interrupts are masked, and that `temp` is declared as a global integer.

Listing 2.2  Non-reentrant function.

```c
int swap()
{
    int x, y;
    Temp = x;
    x = y;
    y = Temp;
}
```

The programme intended to make `swap()` usable by any task. Figure 2.6 shows what could happen if a low-priority task is interrupted while `swap()` is executing. Note that at this point `temp` contains 1. The ISR makes the higher-priority task ready to run, so at the completion of the ISR (F2.6(c)), the kernel (assuming µC/OS-II) is invoked to switch to this task (F2.6(a)). The high-priority task sets `temp` to 3 and swaps the contents of its variables correctly, i.e., `x` is 4 and `y` is 3. The high-priority task eventually relinquishes control to the low-priority task (F2.6(b)) by calling a kernel service to delay itself for one clock tick (described later). The lower-priority task is thus resumed (F2.6(b)). Note that at this point, `temp` is still set to 1! When the low-priority task resumes execution, it sets `y` to 3 instead of 1.

Note that this is a simple example, so it is obvious how to make the code reentrant. However, other situations are not as easy to solve. An error caused by a non-reentrant function may not show up in your application during the testing phase; it will most likely occur once the product has been delivered! If you are new to multitasking, you will need to be careful when using non-reentrant functions.

You can make `swap()` reentrant with one of the following techniques:

- Declare `temp` local to `swap()`.
- Disable interrupts before the operation and enable them afterwards.
- Use a semaphore (described later).
Figure 2.6  Non-reentrant function.

LOW-PRIORITY TASK

\[\text{while (1) \{} \]
\[\text{x = 1;} \]
\[\text{y = 2;} \]
\[\text{swap (x, y);} \]
\[\text{Temp = y;} \]
\[\text{ISR} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{Temp = y;} \]
\[\text{y = Temp;} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{Temp = 3;} \]
\[\text{\}} \]
\[\text{Temp = 3;} \]

HIGH-PRIORITY TASK

\[\text{while (1) \{} \]
\[\text{x = 3;} \]
\[\text{t = 4;} \]
\[\text{ISR} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{Temp = y;} \]
\[\text{y = Temp;} \]
\[\text{O.S.} \]
\[\text{O.S.} \]
\[\text{Temp = 3;} \]
\[\text{\}} \]

If the interrupt occurs either before or after \text{swap()}, the \text{x} and \text{y} values for both tasks will be correct.

2.12 Round-Robin Scheduling

When two or more tasks have the same priority, the kernel allows one task to run for a predetermined amount of time, called a quantum. Then selects another task. This is also called nice slicing. The kernel gives control to the next task in line if:
- the current task has no work to do during its time slice or
- the current task completes before the end of its time slice.

\text{\mu}COS-II does not currently support round-robin scheduling. Each task must have a unique priority in your application.

2.13 Task Priority

A priority is assigned to each task. The more important the task, the higher the priority given to it.

2.14 Static Priorities

Task priorities are said to be static when the priority of each task does not change during the application's execution. Each task is given a fixed priority at compile time. All task and their timing constraints are known at compile time in a system where priorities are static.
2.15 Dynamic Priorities

Task priorities are said to be dynamic if the priority of tasks can be changed during the application’s execution; each task can change its priority at run time. This is a desirable feature to have in a real-time kernel to avoid priority inversions.

2.16 Priority Inversions

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel. Figure 2.7 illustrates a priority inversion scenario. Task 1 has a higher priority than Task 2, which in turn has a higher priority than Task 3. Task 1 and Task 2 are both waiting for an event to occur and Task 3 is executing [P2.7(1)]. At some point, Task 3 acquires a semaphore (see section 2.18.04, Semaphores), which it needs before it can access a shared resource [P2.7(2)]. Task 3 performs some operations on the acquired resource [P2.7(3)] until it is preempted by the high-priority task, Task 1 [P2.7(4)]. Task 1 executes for a while until it also wants to access the resource [P2.7(5)]. Because Task 3 owns the resource, Task 1 has to wait until Task 3 releases the semaphore. As Task 1 tries to get the semaphore, the kernel notes that the semaphore is already owned; thus, Task 1 is suspended and Task 3 is resumed [P2.7(6)]. Task 3 continues execution until it is preempted by Task 2 because the event that Task 2 was waiting for occurred [P2.7(7)]. Task 2 handles the event [P2.7(8)] and when it’s done, Task 2 relinquishes the CPU back to Task 3 [P2.7(9)]. Task 3 finishes working with the resource [P2.7(10)] and releases the semaphore [P2.7(11)]. At this point, the kernel knows that a higher priority task is waiting for the semaphore, and a context switch is done to resume Task 1. At this point, Task 1 has the semaphore and can access the shared resource [P2.7(12)].

The priority of Task 1 has been virtually reduced to that of Task 3 because it was waiting for the resource that Task 3 owned. The situation was aggravated when Task 2 preempted Task 3, which further delayed the execution of Task 1.

You can correct this situation by raising the priority of Task 3, just for the time it takes to access the resource, then restoring the original priority level when the task is finished. The priority of Task 3 must be raised up to or above the highest priority of the other tasks competing for the resource. A multitasking kernel should allow task priorities to change dynamically to help prevent priority inversions. However, it takes some time to change a task’s priority. What if Task 3 had completed access of the resource before it was preempted by Task 1 and then by Task 2? Had you raised the priority of Task 3 before accessing the resource and then lowered it back when done, you would have wasted valuable CPU time. What is really needed to avoid priority inversion is a kernel that changes the priority of a task automatically. This is called priority inheritance, which μC/OS-II unfortunately does not support. There are, however, some commercial kernels that do.
Figure 2.7 Priority inversion problem.

Figure 2.7 illustrates what happens when a kernel supports priority inheritance. As with the previous example, Task 3 is running [F2.8(1)] and acquires a semaphore to access a shared resource [F2.8(2)]. Task 3 accesses the resource [F2.8(3)] and then is preempted by Task 1 [F2.8(4)]. Task 1 executes [F2.8(5)] and tries to obtain the semaphore [F2.8(6)]. The kernel sees that Task 3 has the semaphore but has a lower priority than Task 1. In this case, the kernel raises the priority of Task 3 to the same level as Task 1. The kernel then switches back to Task 3 so that this task can continue with the resource [F2.8(7)]. When Task 3 is done with the resource, it releases the semaphore [F2.8(8)]. At this point, the kernel reduces the priority of Task 3 to its original value and gives the semaphore to Task 1 which is now free to continue [F2.8(9)]. When Task 1 is done executing [F2.8(10)], the medium-priority task (i.e., Task 2) gets the CPU [F2.8(11)]. Note that Task 2 could have been ready to run any time between F2.8(3) and (10) without affecting the outcome. There is still some level of priority inversion that cannot be avoided.
Figure 2.8  Kernel that supports priority inheritance.

Task 1 (H)
Task 2 (M)
Task 3 (L)
Task 3 Goes Semaphore
(2)
Task 1 Preempts Task 3
(4)
Task 1 Tries to get Semaphore
(Priority of Task 3 is raised to Task 1's)
(6)
Task 1 Completes
(10)
Task 3 Releases the Semaphore
(Task 1 Resumes)
(8)

2.17 Assigning Task Priorities

Assigning task priorities is not a trivial undertaking because of the complex nature of real-time systems. In most systems, not all tasks are considered critical. Noncritical tasks should obviously be given low priorities. Most real-time systems have a combination of SOFT and HARD requisitons. In a SOFT real-time system, tasks are performed as quickly as possible, but they don't have to finish by specific times. In HARD real-time systems, tasks have to be performed not only correctly, but on time.
An interesting technique called Rate Monotonic Scheduling (RMS) has been established to assign task priorities based on how often tasks execute. Simply put, tasks with the highest rate of execution are given the highest priority (Figure 2.9).

**Figure 2.9 Assigning task priorities based on task execution rate.**

RMS makes a number of assumptions:

- All tasks are periodic (they occur at regular intervals).
- Tasks do not synchronize with one another, share resources, or exchange data.
- The CPU must always execute the highest priority task that is ready to run. In other words, pre-emptive scheduling must be used.

Given a set of tasks that are assigned RMS priorities, the basic RMS theorem states that all task HARD real-time deadlines will always be met if the inequality in Equation (2.1) is verified.

\[ \Sigma \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \]

where, \(C_i\) corresponds to the maximum execution time of task \(i\) and \(T_i\) corresponds to the execution period of task \(i\). In other words, \(C_i/T_i\) corresponds to the fraction of CPU time required to execute task \(i\). Table 2.1 shows the value for sizes \(n(2^{1/n} - 1)\) based on the number of tasks. The upper bound for an infinite number of tasks is given by \(\ln(2)\) or 0.693. This means that to meet all HARD real-time deadlines based on RMS, CPU utilization of all time-critical tasks should be less than 70 percent! Note that you can still have non-time-critical tasks in a system and thus use 100 percent of the CPU's time. Using 100 percent of your CPU's time is not a desirable goal because it does not allow for code changes and added features. As a rule of thumb, you should always design a system to use less than 60 to 70 percent of your CPU.

RMS says that the highest rate task has the highest priority. In some cases, the highest rate task may not be the most important task. Your application will thus dictate how you need to assign priorities. However, RMS is an interesting starting point.
Table 2.1  Allowable CPU utilization based on number of tasks.

<table>
<thead>
<tr>
<th>Number of Tasks</th>
<th>( n(2^{1/n} - 1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>0.828</td>
</tr>
<tr>
<td>3</td>
<td>0.779</td>
</tr>
<tr>
<td>4</td>
<td>0.756</td>
</tr>
<tr>
<td>5</td>
<td>0.743</td>
</tr>
<tr>
<td>( \infty )</td>
<td>0.693</td>
</tr>
</tbody>
</table>

2.18 Mutual Exclusion

The easiest way for tasks to communicate with each other is through shared data structures. This is especially easy when all tasks exist in a single address space and can reference global variables, pointers, buffers, linked lists, ring buffers, etc. Although sharing data simplifies the exchange of information, you must ensure that each task has exclusive access to the data to avoid contention and data corruption. The most common methods of obtaining exclusive access to shared resources are:

- disabling interrupts,
- performing test-and-set operations,
- disabling scheduling, and
- using semaphores.

2.18.01 Disabling and Enabling Interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts, as shown in the pseudocode in Listing 2.3.

Listing 2.3  Disabling and enabling interrupts.

```
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

\( \mu\text{COS-II} \) uses this technique (as do most, if not all, kernels) to access internal variables and data structures. In fact, \( \mu\text{COS-II} \) provides two macros that allow you to disable and then enable interrupts from your C code: \texttt{OS_ENTER_CRITICAL()} and \texttt{OS_EXIT_CRITICAL()}, respectively. You need to use these macros in tandem, as shown in Listing 2.4.
Listing 2.4 Using \mu C/OS-II macros to disable and enable interrupts.

```c
void function (void)
{
    OS_ENTER_CRITICAL();
    /* You can access shared data here */
    OS_EXIT_CRITICAL();
}
```

You must be careful, however, not to disable interrupts for too long because this affects the response of your system to interrupts. This is known as interrupt latency. You should consider this method when you are changing or copying a few variables. Also, this is the only way that a task can share variables or data structures with an ISR. In all cases, you should keep interrupts disabled for as little time as possible.

If you use a kernel, you are basically allowed to disable interrupts for as much time as the kernel does without affecting interrupt latency. Obviously, you need to know how long the kernel will disable interrupts. Any good kernel vendor will provide you with this information. After all, if they sell a real-time kernel, time is important!

2.18.02 Test-And-Set

If you are not using a kernel, two functions could 'agree' that to access a resource, they must check a global variable and if the variable is 0, the function has access to the resource. To prevent the other function from accessing the resource, however, the first function that gets the resource simply sets the variable to 1. This is commonly called a Test-And-Set (or TAS) operation. Either the TAS operation must be performed indivisibly (by the processor) or you must disable interrupts when doing the TAS on the variable, as shown in Listing 2.5.

Listing 2.5 Using Test-And-Set to access a resource.

```c
Disable interrupts;
if ('Access Variable' is 0) {
    Set variable to 1;
    Reenable interrupts;
    Disable interrupts;
    Set the 'Access Variable' back to 0;
    Reenable interrupts;
} else {
    Reenable interrupts;
    /* You don't have access to the resource, try back later; */
}
```
Some processors actually implement a TAS operation in hardware (e.g., the 8000 family of processors have the TAS instruction).

### 2.18.03 Disabling and Enabling the Scheduler

If your task is not sharing variables or data structures with an ISR, you can disable and enable scheduling, as shown in Listing 2.6 (using &WOS-II as an example). In this case, two or more tasks can share data without the possibility of contention. You should note that while the scheduler is locked, interrupts are disabled, and if an interrupt occurs while in the critical section, the ISR is executed immediately. At the end of the ISR, the kernel always returns to the interrupted task, even if a higher priority task has been made ready to run by the ISR. The scheduler is invoked when OS_schedLock() is called to see if a higher priority task has been made ready to run by the task or an ISR. A context switch results if a higher priority task is ready to run. Although this method works well, you should avoid disabling the scheduler because it defeats the purpose of having a kernel in the first place. The next method should be chosen instead.

**Listing 2.6 Accessing shared data by disabling and enabling scheduling.**

```c
void Function (void)
{
    OS_schedLock();
    // You can access shared data in here (interrupts are recognized)
    OS_schedUnlock();
}
```

### 2.18.04 Semaphores

The semaphore was invented by Edsger Dijkstra in the mid-1960s. It is a protocol mechanism offered by multitasking kernels. Semaphores are used to:

- control access to a shared resource (mutual exclusion),
- signal the occurrence of an event, and
- allow two tasks to synchronize their activities.

A semaphore is a key that your code acquires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner. In other words, the requesting task says: "Give me the key. If someone else is using it, I am willing to wait for it!"

There are two types of semaphores: binary semaphores and counting semaphores. As its name implies, a binary semaphore can only take two values: 0 or 1. A counting semaphore allows values between 0 and 255, 65535, or 4294967295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. The actual size depends on the kernel used. Along with the semaphore's value, the kernel also needs to keep track of tasks waiting for the semaphore's availability.

Generally, only three operations can be performed on a semaphore: INITIALIZE (also called CREATE), WAIT (also called PEND), and SIGNAL (also called POST). The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty.
A task desiring the semaphore will perform a \texttt{WAIT} operation. If the semaphore is available (the semaphore value is greater than \texttt{0}), the semaphore value is decremented and the task continues execution. If the semaphore’s value is \texttt{0}, the task performing a \texttt{WAIT} or the semaphore is placed in a waiting list. Most kernels allow you to specify a timeout, if the semaphore is not available within a certain amount of time, the requesting task is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller.

A task releases a semaphore by performing a \texttt{SIGNAL} operation. If no task is waiting for the semaphore, the semaphore value is simply incremented. If any task is waiting for the semaphore, however, one of the tasks is made ready to run and the semaphore value is not incremented; the key is given to one of the tasks waiting for it. Depending on the kernel, the task that receives the semaphore is either

- the highest priority task waiting for the semaphore or
- the first task that requested the semaphore (First In First Out, or \texttt{FIPO}).

Some kernels have an option that allows you to choose either method when the semaphore is initialized. \texttt{uCOS-II} only supports the first method. If the real-time task has a higher priority than the current task (the task releasing the semaphore), a context switch occurs (with a protective kernel) and the higher priority task resumes execution; the current task is suspended until it again becomes the highest priority task ready to run.

Listing 2.7 shows how you can share data using a semaphore (in \texttt{uCOS-II}). Any task needing access to the same shared data calls \texttt{OSSemPend()}, and when the task is done with the data, the task calls \texttt{OSSemPost()}. Both of these functions are described later. You should note that a semaphore is an object that needs to be initialized before it is used; for mutual exclusion, a semaphore is initialized to \texttt{0}. Using a semaphore to access shared data doesn’t affect interrupt latency. If an ISR or the execution of the task makes a higher priority task ready to run while accessing shared data, the higher priority task executes immediately.

\textbf{Listing 2.7 Accessing shared data by obtaining a semaphore.}

```
# ifndef _SHARED_DATA

void Function (void)
{

    /* You can access shared data in here (interruption are recognized) */

    OSSemPost (hSharedDataSem);
}
```

Semaphores are especially useful when tasks share I/O devices. Imagine what would happen if two tasks were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each task. For instance, the printout from Task 1 printing "I am Task 1!" and Task 2 printing "I am Task 2!" could result in:

```
I am mum T Tansak kI 12!
```
In this case, use a semaphore and initialize it to 1 (i.e., a binary semaphore). The rule is simple: to access the printer each task first must obtain the resource's semaphore. Figure 2.10 shows tasks competing for a semaphore to gain exclusive access to the printer. Note that the semaphore is represented symbolically by a key, indicating that each task must obtain this key to use the printer.

**Figure 2.10 Using a semaphore to get permission to access a printer.**

The above example implies that each task must know about the existence of the semaphore in order to access the resource. There are situations when it is better to encapsulate the semaphore. Each task would thus not know that it is actually acquiring a semaphore when accessing the resource. For example, in RS-232C port is used by multiple tasks to send commands and receive responses from a device connected at the other end (Figure 2.11).

The function `CommSendCmd()` is called with three arguments: the ASCII string containing the command, a pointer to the response string from the device, and finally, a timeout in case the device doesn't respond within a certain amount of time. The pseudocode for this function is shown in Listing 2.8.

**Listing 2.8 Encapsulating a semaphore.**

```c
INMUX CommSendCmd(char *cmd, char *response, INT16U timeout) {
    Acquire port's semaphore;
    Send command to device;
    Wait for response [with timeout];
    if (timeout out) {
        Release semaphore;
        return (error code);
    } else {
```

Listing 2.8 Encapsulating a semaphore. (Continued)

    release semaphore;
    return (no error);
}

Each task that needs to send a command to the device has to call this function. The semaphore is assumed to be initialized to 1 (i.e., available) by the communication driver initialization routine. The first task that calls CommandSendCmd() acquires the semaphore, proceeds to send the command, and waits for a response. If another task attempts to send a command while the port is busy, this second task is suspended until the semaphore is released. The second task appears simply to have made a call to a normal function that will not return until the function has performed its duty. When the semaphore is released by the first task, the second task acquires the semaphore and is allowed to use the RS-232C port.

Figure 2.11 Hiding a semaphore from tasks.

A counting semaphore is used when a resource can be used by more than one task at the same time. For example, a counting semaphore is used in the management of a buffer pool as shown in Figure 2.12. Assume that the buffer pool initially contains 10 buffers. A task would obtain a buffer from the buffer manager by calling BufReq(). When the buffer is no longer needed, the task would return the buffer to the buffer manager by calling BufRel(). The pseudocode for these functions is shown in Listing 2.9.
Listing 2.9  Buffer management using a semaphore.

```c
HWF *bufHand(void)
{
    HWF *ptr;

    Acquire a semaphore;
    Disable interrupts;
    ptr = BufFreedList;
    BufFreedlist = ptr->BufNext;
    Enable interrupts;
    return (ptr);
}

void BufRel(HWF *ptr)
{
    Disable interrupts;
    ptr->BufNext = BufFreedlist;
    BufFreedlist = ptr;
    Enable interrupts;
    Release semaphore;
}
```

Figure 2.12 Using a counting semaphore.
The buffer manager will satisfy the first 10 buffer requests because there are 10 keys. When all semaphores are used, a task requesting a buffer is suspended until a semaphore becomes available. Interrupts are disabled to gain exclusive access to the linked list (this operation is very quick). When a task is finished with the buffer it acquired, it calls Buffered[] to return the buffer to the buffer manager; the buffer is inserted into the linked list before the semaphore is released. By encapsulating the interface to the buffer manager in BufReq[] and BufRel[], the caller doesn't need to be concerned with the actual implementation details.

Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in most situations. The overhead involved in acquiring and releasing the semaphore can consume valuable time. You can do the job just as efficiently by disabling and enabling interrupts (see section 2.18.1), Disabling and Enabling Interrupts). Suppose that two tasks are sharing a 32-bit integer variable. The first task increments the variable while the other task clears it. If you consider how long a processor takes to perform either operation, you will realize that you do not need a semaphore to gain exclusive access to the variable. Each task simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete. A semaphore should be used, however, if the variable is a floating-point variable and the microprocessor doesn't support floating-point instructions in hardware. In this case, the processing time involved in processing the floating-point variable could have affected interrupt latency if you had disabled interrupts.

2.19 Deadlock (or Deadly Embrace)

A deadlock, also called a deadly embrace, is a situation in which two tasks are each unknowingly waiting for resources held by the other. Assume task T1 has exclusive access to resource R1 and task T2 has exclusive access to resource R2. If T1 needs exclusive access to R2 and T2 needs exclusive access to R1, neither task can continue. They are deadlocked. The simplest way to avoid a deadlock is for tasks to
- acquire all resources before proceeding,
- release the resources in the same order, and
- release the resources in the reverse order.

Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken. If the semaphore is not available within a certain amount of time, the task requesting the resource resumes execution. Some form of error code must be returned to the task to notify it that a timeout occurred. A return error code prevents the task from thinking it has obtained the resource. Deadlocks generally occur in large multitasking systems, not in embedded systems.

2.20 Synchronization

A task can be synchronized with an ISR (or another task when no data is being exchanged) by using a semaphore as shown in Figure 2.13. Note that, in this case, the semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key). When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for this type of synchronization is called a unilateral rendezvous. A task initiates an I/O operation and waits for the semaphore. When the I/O operation is complete, an ISR (or another task) signals the semaphore and the task is resumed.
Figure 2.13  Synchronizing tasks and ISRs.

If the kernel supports counting semaphores, the semaphore would accumulate events that have not yet been processed. Note that more than one task can be waiting for an event to occur. In this case, the kernel could signal the occurrence of the event either to
- the highest priority task waiting for the event to occur or
- the first task waiting for the event.

Depending on the application, more than one ISR or task could signal the occurrence of the event.

Two tasks can synchronize their activities by using two semaphores, as shown in Figure 2.14. This is called a bilateral rendezvous. A bilateral rendezvous is similar to a unilateral rendezvous, except both tasks must synchronize with one another before proceeding.

For example, two tasks are executing as shown in Listing 2.10. When the first task reaches a certain point, it signals the second task (L2.10(1)) then waits for a return signal (L2.10(2)). Similarly, when the second task reaches a certain point, it signals the first task (L2.10(3)) and waits for a return signal (L2.10(4)). At this point, both tasks are synchronized with each other. A bilateral rendezvous cannot be performed between a task and an ISR because an ISR cannot wait on a semaphore.

Figure 2.14  Tasks synchronizing their activities.
Listing 2.10 Bilateral rendezvous.

Task1()
{
  for (;;) {
    Perform operation;
    Signal task #2;  (1)
    Wait for signal from task #1;  (2)
    Continue operation;
  }
}

Task2()
{
  for (;;) {
    Perform operation;
    Signal task #1;  (3)
    Wait for signal from task #2;  (4)
    Continue operation;
  }
}

2.21 Event Flags

Event flags are used when a task needs to synchronize with the occurrence of multiple events. The task can be synchronized when any of the events have occurred. This is called disjunctive synchronization (logical OR). A task can also be synchronized when all events have occurred. This is called conjunctive synchronization (logical AND). Disjunctive and conjunctive synchronization are shown in Figure 2.15.

Common events can be used to signal multiple tasks, as shown in Figure 2.16. Events are typically grouped. Depending on the kernel, a group consists of 8, 16, or 32 events, each represented by a bit (usually 32 bits, though). Tasks and ISRs can set or clear any event in a group. A task is resumed when all the events it requires are satisfied. The evaluation of which task will be resumed is performed when a new set of events occurs (i.e., during a SET operation).

Kernels supporting event flags often serve to SET event flags, CLEAR event flags, and WAIT for event flags (conjunctively or disjunctively). μC/OS-II does not currently support event flags.
### Figure 2.15  Disjunctive and conjunctive synchronization.

![Diagram](image)

#### DISJUNCTIVE SYNCHRONIZATION

- **Events** → OR → Semaphore → POST → TASK → PEND

#### CONJUNCTIVE SYNCHRONIZATION

- **Events** → AND → Semaphore → POST → TASK → PEND

---

### 2.22 Intertask Communication

It is sometimes necessary for a task or an ISR to communicate information to another task. This information transfer is called **intertask communication**. Information may be communicated between tasks in two ways: through global data or by sending messages.

When using global variables, each task or ISR must ensure that it has exclusive access to the variables. If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts. If two tasks are sharing data, each can gain exclusive access to the variables either by disabling and enabling interrupts or with the use of a semaphore (as we have seen). Note that a task can only communicate information to an ISR by using global variables. A task is not aware when a global variable is changed by an ISR, unless the ISR signals the task by using a semaphore or unless the task polls the contents of the variable periodically. To correct this situation, you should consider using either a message mailbox or a message queue.
2.23 Message Mailboxes

Messages can be sent to a task through kernel services. A Message Mailbox, also called a message exchange, is typically a pointer-size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more tasks can receive messages through a service provided by the kernel. Both the sending task and receiving task agree on what the pointer is actually pointing to.

A waiting list is associated with each mailbox in case more than one task wants to receive messages through the mailbox. A task desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run and an error code (indicating that a timeout has occurred) is returned to it. When a message is deposited into the mailbox, either the highest priority task waiting for the message is given the message (priority based) or the first task to request a message is given the message (First-In-First-Out, or FIFO). Figure 2.17 shows a task depositing a message into a mailbox. Note that the mailbox is represented by an L-beam and the timeout is represented by an hourglass. The number next to the hourglass represents the number of clock ticks (described later) the task will wait for a message to arrive.
Kernels typically provide the following mailbox services.

- Initialize the contents of a mailbox. The mailbox initially may or may not contain a message.
- Deposit a message into the mailbox (POST).
- Wait for a message to be deposited into the mailbox (PEND).
- Get a message from a mailbox if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox. A return code is used to notify the caller about the outcome of the call.

Message mailboxes can also simulate binary semaphores. A message in the mailbox indicates that the resource is available, and an empty mailbox indicates that the resource is already in use by another task.

**Figure 2.17 Message mailbox.**

---

### 2.24 Message Queues

A message queue is used to send one or more messages to a task. A message queue is basically an array of mailboxes. Through a service provided by the kernel, a task or an ISR can deposit a message (write pointer) into a message queue. Similarly, one or more tasks can receive messages through a service provided by the kernel. Both the sending task and receiving task agree as to what the pointer is actually pointing to. Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO). In addition, to extract messages in a FIFO fashion, μC/OS-II allows a task to get messages Last-In-First-Out (LIFO).

As with the mailbox, a waiting list is associated with each message queue. In case more than one task is to receive messages through the queue, a task desiring a message from an empty queue is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run and an error code (indicating a timeout has occurred) is returned to it. When a message is deposited into the queue, either the highest priority task or the first task to wait for the message is given the message. **Figure 2.18 shows an ISR (Interrupt Service Routine) depositing a message into a queue. Note that the queue is represented graphically by a double I-beam. The “10” indicates the number of messages that can accumulate in the queue. A “0” next to the hourglass indicates that the task will wait forever for a message to arrive.”
Kernels typically provide the message queue services listed below.

- Initialize the queue. The queue is always assumed to be empty after initialization.
- Deposit a message into the queue (POST).
- Wait for a message to be deposited into the queue (PEND).
- Get a message from a queue if one is present, but do not suspend the caller if the queue is empty (ACCEPT). If the queue contains a message, the message is extracted from the queue. A return code is used to notify the caller about the outcome of the call.

**Figure 2.18 Message queue.**

![Diagram showing message queue process](image)

### 2.25 Interrupts

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves part (or all) of its context (i.e., registers) and jumps to a special subroutine called an *Interrupt Service Routine*, or ISR. The ISR processes the event, and upon completion of the ISR, the program returns to

- the background for a foreground/background system,
- the interrupted task for a non-preemptive kernel, or
- the highest priority task ready to run for a preemptive kernel.

Interrupts allow a microprocessor to process events when they occur. This prevents the microprocessor from continuously polling an event to see if it has occurred. Microprocessors allow interrupts to be ignored and recognized through the use of two special instructions: *disable interrupts and enable interrupts*, respectively. In a real-time environment, interrupts should be disabled as little as possible. Existing interrupts affect interrupt latency (see section 2.26, Interrupt Latency) and may cause interrupts to be missed. Processors generally allow interrupts to be reset. This means that while servicing an interrupt, the processor will recognize and service other (more important) interrupts, as shown in Figure 2.19.

### 2.26 Interrupt Latency

Probably the most important specification of a real-time kernel is the amount of time interrupts are disabled. All real-time systems disable interrupts to manipulate critical sections of code and reenable interrupts when the critical section has executed. The longer interrupts are disabled, the higher the *interrupt latency*. Interrupt latency is given by Equation [2.2].

\[
\text{Maximum amount of time interrupts are disabled} = \text{Time to start executing the first instruction in the ISR}
\]
2.27 Interrupt Response

Interrupt response is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt. The interrupt response time accounts for all the overhead involved in handling an interrupt. Typically, the processor’s context (CPU registers) is saved on the stack before the user code is executed.

For a foreground/background system, the user ISR code is executed immediately after saving the processor’s context. The response time is given by Equation (2.3).

\[ \text{Interrupt latency} + \text{Time to save the CPU's context} \]  

For a non-preemptive kernel, the user ISR code is executed immediately after the processor’s context is saved. The response time to an interrupt for a non-preemptive kernel is given by Equation (2.4).

\[ \text{Interrupt latency} + \text{Time to save the CPU's context} \]  

For a preemptive kernel, a special function provided by the kernel needs to be called. This function notifies the kernel that an ISR is in progress and allows the kernel to keep track of interrupt nesting. For
\( \mu \text{COS-II}, \) this function is called `OSIntEnter()`. The response time to an interrupt for a preemptive kernel is given by Equation (2.5).

\[ \text{[2.5]} \]

- **Interrupt latency**
  - Time to save the CPU's context
  - Execution time of the kernel ISR entry function

A system’s worst case interrupt response time is its only response. Your system may respond to interrupts in 50\(\mu\)s 99 percent of the time, but if it responds to interrupts in 250\(\mu\)s the other 1 percent, you must assume a 250\(\mu\)s interrupt response time.

### 2.28 Interrupt Recovery

Interrupt recovery is defined as the time required for the processor to return to the interrupted code. Interrupt recovery in a foreground/background system simply involves restoring the processor’s context and returning to the interrupted task. Interrupt recovery is given by Equation (2.6).

\[ \text{[2.6]} \]

- Time to restore the CPU’s context
- Time to execute the return from interrupt instruction

As with a foreground/background system, interrupt recovery with a non-preemptive kernel (Equation (2.7)) simply involves restoring the processor’s context and returning to the interrupted task.

\[ \text{[2.7]} \]

- Time to restore the CPU’s context
- Time to execute the return from interrupt instruction

For a preemptive kernel, interrupt recovery is more complex. Typically, a function provided by the kernel is called at the end of the ISR. For \( \mu \text{COS-II}, \) this function is called `OSIntExit()` and allows the kernel to determine if all interrupts have nested. If they have nested (i.e., a return from interrupt would return to task-level code), the kernel determines its highest priority task has been made ready to run as a result of the ISR. If a higher priority task is ready to run as a result of the ISR, this task is resumed. Note that, in this case, the interrupted task will resume only when it again becomes the highest priority task ready to run. For a preemptive kernel, interrupt recovery is given by Equation (2.8).

\[ \text{[2.8]} \]

- Time to determine if a higher priority task is ready
- Time to restore the CPU’s context of the highest priority task
- Time to execute the return from interrupt instruction

### 2.29 Interrupt Latency, Response, and Recovery

Figures 2.20 through 2.22 show the interrupt latency, response, and recovery for a foreground/background system, a non-preemptive kernel, and a preemptive kernel, respectively.

You should note that for a preemptive kernel, the exit function either decides to return to the interrupted task (F2.22A) or to a higher priority task that the ISR has made ready to run (F2.22B). In the latter case, the execution time is slightly longer because the kernel has to perform a context switch. I made the difference in execution time somewhat to scale assuming \( \mu \text{COS-II} \) on an Intel 80186 processor (see Table 9.3). Execution times of \( \mu \text{COS-II} \) services on 33MHz 80186). This allows you to see the cost (in execution time) of switching context.
2.30 ISR Processing Time

Although ISRs should be as short as possible, there are no absolute limits on the amount of time for an ISR. One cannot say that an ISR must always be less than 100μs, 500μs, or 1ms. If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be. In most cases, however, the ISR should recognize the interrupt, obtain data or a status from the interrupting device, and signal a task to perform the actual processing. You should also consider whether the overhead involved in signaling a task is more than the processing of the interrupt. Signaling a task from an ISR (i.e. through a semaphore, a mailbox, or a queue) requires some processing time. If processing your interrupt requires less than the time required to signal a task, you should consider processing the interrupt in the ISR itself and possibly enabling interrupts to allow higher priority interrupts to be recognized and serviced.

**Figure 2.20** Interrupt latency, response, and recovery (foreground/background).

```
<table>
<thead>
<tr>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Diagram showing ISR processing time]</td>
</tr>
</tbody>
</table>
```

2.31 Nonmaskable Interrupts (NMIs)

Sometimes, an interrupt must be serviced as quickly as possible and cannot afford to have the latency imposed by a kernel. In these situations, you may be able to use the **Nonmaskable Interrupt (NMI)** provided on most microprocessors. Because the NMI cannot be disabled, interrupt latency, response, and recovery are minimal. The NMI is generally reserved for drastic measures such as saving important
information during a power down. If, however, your application doesn’t have this requirement, you could use the NMI to service your most time-critical ISR. The following equations show how to determine the interrupt latency [2.9], response [2.10], and recovery [2.11], respectively, of an NMI.

\begin{align*}
[2.9] & \quad \text{Time to execute longest instruction} + \text{Time to start executing the NMI ISR} \\
[2.10] & \quad \text{Interrupt latency} + \text{Time to save the CPU's context} \\
[2.11] & \quad \text{Time to restore the CPU's context} + \text{Time to execute the return from interrupt instruction}
\end{align*}

I have used the NMI in an application to respond to an interrupt that could occur every 150µs. The processing time of the ISR took from 80 to 125µs, and the kernel I used disabled interrupts for about 45µs. As you can see, if I had used maskable interrupts, the ISR could have been late by 20µs.

When you are servicing an NMI, you cannot use kernel services to signal a task because NMIs cannot be disabled to access critical sections of code. However, you can still pass parameters to and from the NMI. Parameters passed must be global variables and the size of these variables must be read or written indivisibly; that is, not as separate byte read or write instructions.

**Figure 2.21 Interrupt latency, response, and recovery (non-preemptive kernel).**
Figure 2.22  Interrupt latency, response, and recovery (preemptive kernel).

TIME

Interrupt Request

TASK

ISR

Interrupt Latency

Kernel's ISR Entry function

CPU context saved

Kernel's ISR Exit function

User ISR Code

Interrupt Response

Interrupt Recovery

A

CPU context restored

B

CPU context restored

TASK

NMIs can be disabled by adding external circuitry, as shown in Figure 2.23. Assuming that both the
interrupt and the NMI are positive-going signals, a simple AND gate is inserted between the interrupt
source and the processor’s NMI input. Interrupts are disabled by writing a 0 to an output port. You
wouldn’t want to disable interrupts to use kernel services, but you could use this feature to pass param-
teors (i.e., larger variables) to and from the ISR and a task.

Figure 2.23  Disabling nonmaskable interrupts.

NMI Interrupt Source

Output Port

To Processor’s NMI Input
Now, suppose that the NMI service routine needs to signal a task every 40 times it executes. If the NMI occurs every 150μs, a signal would be required every 6ms (40 x 150μs). From a NMI ISR, you cannot use the kernel to signal the task, but you could use the scheme shown in Figure 2.24. In this case, the NMI service routine would generate a hardware interrupt through an output port (i.e., bring an output high). Since the NMI service routine typically has the highest priority and interrupt nesting is typically not allowed while servicing the NMI ISR, the interrupt would not be recognized until the end of the NMI service routine. At the completion of the NMI service routine, the processor would be interrupted to service this hardware interrupt. This ISR would clear the interrupt source (i.e., bring the port output low) and post to a semaphore that would wake up the task. As long as the task services the semaphore well within 6ms, your deadline would be met.

**Figure 2.24 Signaling a task from a nonmaskable interrupt.**

![Diagram showing signaling a task from a nonmaskable interrupt](image)

### 2.32 Clock Tick

A **clock tick** is a special interrupt that occurs periodically. This interrupt can be viewed as the system's heartbeat. The time between interrupts is application specific and is generally between 10 and 200ms. The clock tick interrupt allows a kernel to delay tasks for an integral number of clock ticks and to provide timeouts when tasks are waiting for events to occur. The faster the tick rate, the higher the overhead imposed on the system.

All kernels allow tasks to be delayed for a certain number of clock ticks. The resolution of delayed tasks is one clock tick; however, this does not mean that its accuracy is one clock tick.

Figures 2.25 through 2.27 are timing diagrams showing a task delaying itself for one clock tick. The shaded areas indicate the execution time for each operation being performed. Note that the time for each operation varies to reflect typical processing, which would include loops and conditional statements (i.e., ifelsenw switch, and ?). The processing time of the Tick ISR has been exaggerated to show that it too is subject to varying execution times.

Case 1 (Figure 2.25) shows a situation where higher priority tasks and ISRs execute prior to the task, which needs to delay for one tick. As you can see, the task attempts to delay for 20ms but because of its priority, actually executes at varying intervals. This causes the execution of the task to jitter.
Figure 2.25 Delaying a task for one tick (Case 1).

Tick Interrupt

Tick ISR

All higher priority tasks

Call to delay 1 tick (20ms)

Call to delay 1 tick (20ms)

Call to delay 1 tick (20ms)

Delayed Task

1

(19ms)

2

(17ms)

3

(27ms)

Case 2 (Figure 2.26) shows a situation where the execution times of all higher priority tasks and ISRs are slightly less than one tick. If the task delays itself just before a clock tick, the task will execute again almost immediately! Because of this, if you need to delay a task at least one clock tick, you must specify one extra tick. In other words, if you need to delay a task for at least five ticks, you must specify six ticks.

Figure 2.26 Delaying a task for one tick (Case 2).

Tick Interrupt

Tick ISR

All higher priority tasks

Call to delay 1 tick (20ms)

Call to delay 1 tick (20ms)

Call to delay 1 tick (20ms)

Delayed Task

1

(6ms)

2

(17ms)

3

(27ms)
Case 3 (Figure 2.27) shows a situation in which the execution times of all higher priority tasks and ISRs extend beyond one clock tick. In this case, the task that tries to delay for one tick actually executes two ticks later and misses its deadline. This might be acceptable in some applications, but in most cases it isn’t.

These situations exist with all real-time kernels. They are related to CPU processing load and possibly incorrect system design. Here are some possible solutions to these problems:

- Increase the clock rate of your microprocessor.
- Increase the time between tick interrupts.
- Rearrange task priorities.
- Avoid using floating-point math (if you must, use single precision).
- Get a compiler that performs better code optimization.
- Write time-critical code in assembly language.
- If possible, upgrade to a faster microprocessor in the same family; that is, 5086 to 40186, 68000 to 68020, etc.

Regardless of what you do, jitter will always occur.

**Figure 2.27 Delaying a task for one tick (Case 3).**

```
Tick Interrupt  --- 20ms ---
Tick ISR
All higher priority tasks
Call to delay 1 tick (20ms)
Delayed Task
Call to delay 1 tick (20ms)  (40ms)
(26ms)
```

2.33 Memory Requirements

If you are designing a foreground/background system, the amount of memory required depends solely on your application code. With a multitasking kernel, things are quite different. To begin with, a kernel requires extra code space (ROM). The size of the kernel depends on many factors. Depending on the features provided by the kernel, you can expect anywhere from 1 to 10000 bytes. A minimal kernel for an 8-bit CPU that provides only scheduling, context switching, semaphore management, delays, and timeouts should require about 1 to 3 kilobytes of code space. The total code space is given by Equation [2.12].

[2.12] Application code size + Kernel code size
Because each task runs independently of the others, it must be provided with its own stack area (RAM). As a designer, you must determine the stack requirement of each task as closely as possible (this is sometimes a difficult undertaking). The stack size must not only account for the task requirements (local variables, function calls, etc.), it must also account for maximum interrupt nesting (saved registers, local storage in ISR(s), etc.). Depending on the target processor and the kernel used, a separate stack can be used to handle all interrupt-level code. This is a desirable feature because the stack requirement for each task can be substantially reduced. Another desirable feature is the ability to specify the stack size of each task on an individual basis (μCOS-II permits this). Conversely, some kernels require that all task stacks be the same size. All kernels require extra RAM to maintain internal variables, data structures, queues, etc. The total RAM required if the kernel does not support a separate interrupt stack is given by Equation [2.13].

\[ \text{(2.13)} \]

\begin{align*}
\text{Application code requirements} & \\
& + \text{Data space (i.e., RAM) needed by the kernel} \\
& + \text{SUM(task stacks)} + \text{MAX(ISR nesting)} \\
\end{align*}

If the kernel supports a separate stack for interrupts, the total RAM required is given by Equation [2.14].

\[ \text{(2.14)} \]

\begin{align*}
\text{Application code requirements} & \\
& + \text{Data space (i.e., RAM) needed by the kernel} \\
& + \text{SUM(task stacks)} + \text{MAX(ISR nesting)} \\
\end{align*}

Unless you have large amounts of RAM to work with, you need to be careful how you use the stack space. To reduce the amount of RAM needed in an application, you must be careful how you use each task’s stack for:

- Large arrays and structures declared locally to functions and ISRs.
- Function (i.e., subroutine) nesting.
- Interrupt nesting.
- Library functions stack usage, and
- Function calls with many arguments.

To summarize, a multitasking system requires more code space (ROM) and data space (RAM) than a foreground/background system. The amount of extra ROM depends only on the size of the kernel, and the amount of RAM depends on the number of tasks in your system.

### 2.34 Advantages and Disadvantages of Real-Time Kernels

A real-time kernel, also called a Real-Time Operating System, or RTOS, allows real-time applications to be designed and expanded easily; functions can be added without requiring major changes to the software. The use of an RTOS simplifies the design process by splitting the application code into separate tasks. With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible. An RTOS allows you to make better use of your resources by providing you with valuable services, such as semaphores, mailboxes, queues, time delays, timeouts, etc.

You should consider using a real-time kernel if your application can afford the extra requirements—extra cost of the kernel, more ROM/RAM, and 2 to 4 percent additional CPU overhead.
The one factor I haven’t mentioned so far is the cost associated with the use of a real-time kernel. In some applications, cost is everything and would preclude you from even considering an RTOS.

There are currently about 80+ RTOS vendors. Products are available for 8-, 16-, 32-, and even 64-bit microprocessors. Some of these packages are complete operating systems and include not only the real-time kernel but also an input/output manager, windowing systems (display), a file system, networking, language interface libraries, debuggers, and cross-platform compilers. The cost of an RTOS varies from $70 to well over $30,000. The RTOS vendor may also require royalties on a per-target-system basis. This is like buying a chip from the RTOS vendor that you include with each unit sold. The RTOS vendors call this silicon software. The royalty fee varies between $5 to about $250 per unit. Like any other software package these days, you also need to consider the maintenance cost, which can set you back another $100 to $5,000 per year!

### 2.35 Real-Time Systems Summary

Table 2.2 summarizes the three types of real-time systems: foreground/background, non-preemptive kernel, and preemptive kernel.

<table>
<thead>
<tr>
<th></th>
<th>Foreground/Background</th>
<th>Non-Preemptive Kernel</th>
<th>Preemptive Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt latency</strong></td>
<td>MAXI (Longest instruction, User int.disable) + Vector to ISR</td>
<td>MAXI (Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR</td>
<td>MAXI (Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR</td>
</tr>
<tr>
<td>(Time)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interrupt response</strong></td>
<td>Int. latency + Save CPU’s context</td>
<td>Int. latency + Save CPU’s context</td>
<td>Interrupt latency + Save CPU’s context + Kernel ISR entry function</td>
</tr>
<tr>
<td>(Time)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interrupt recovery</strong></td>
<td>Restore background’s context + Return from int.</td>
<td>Restore task’s context + Return from int.</td>
<td>Find highest priority task + Return from highest priority task’s context + Return from interrupt</td>
</tr>
<tr>
<td>(Time)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Task response</strong></td>
<td>Background + Find highest priority task + Context switch</td>
<td>Longest task + Find highest priority task’s context</td>
<td>Find highest priority task + Context switch</td>
</tr>
<tr>
<td>(Time)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ROM size</strong></td>
<td>Application code</td>
<td>Application code</td>
<td>Application code</td>
</tr>
<tr>
<td></td>
<td>+ Kernel code</td>
<td>+ Kernel code</td>
<td>+ Kernel code</td>
</tr>
<tr>
<td><strong>RAM size</strong></td>
<td>Application code</td>
<td>Application code</td>
<td>Application code</td>
</tr>
<tr>
<td></td>
<td>+ Kernel RAM</td>
<td>+ Kernel RAM</td>
<td>+ Kernel RAM</td>
</tr>
<tr>
<td></td>
<td>+ SUM(Task stacks)</td>
<td>+ SUM(Task stacks)</td>
<td>+ SUM(Task stacks)</td>
</tr>
<tr>
<td></td>
<td>+ MAX(ISR stack)</td>
<td>+ MAX(ISR stack)</td>
<td>+ MAX(ISR stack)</td>
</tr>
<tr>
<td><strong>Services available?</strong></td>
<td>Application code must</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
2.36 Bibliography


Chapter 3

Keyboards

A large number of embedded products, such as microwave ovens, FAX machines, copiers, laser printers, Point Of Sale (POS) terminals, Programmable Logic Controllers (PLCs), and so on, rely on a keyboard or keypad interface for user input. The keyboard might be used to input numerical data as well as to select the operating mode of the controlling device. As an embedded system designer, you are always concerned with the cost of your products. Chips are currently available to perform keyboard scanning, but a software approach to keyboard scanning has the benefit of reducing the recurring cost of a system and requires very little CPU overhead.

In this chapter, I will describe how a microprocessor can scan a keyboard, and I will also provide you with a complete, portable 8 x 8 matrix keyboard scanning module. The module can scan any keyboard matrix arrangement up to an 8 x 8 matrix, but can easily be modified to handle a large number of keys. The matrix keyboard module code is an important building block for embedded systems. The keyboard module presented in this chapter has the following features:

- Scans any keyboard arrangement from a 3x3 to an 8x8 key matrix.
- Provides buffer (user configurable buffer size).
- Supports auto-repeat.
- Keeps track of how long a key has been pressed.
- Allows up to three Shift keys.

All you need to do to use this module is to wire three simple hardware interface functions and set the value of I7 hom key constants. The keyboard module assumes the presence of a real-time kernel but can easily be modified to work in a foreground/background environment.

3.00 Keyboard Basics

A momentary-contact switch is typically used in a keyboard, and a closure can easily be detected by a microprocessor using the simple circuit shown in Figure 3.1. The pull-up resistor provides a logic 1 when the switch is opened and a logic 0 when the switch is closed. Unfortunately, switches are not perfect in that they do not generate a crisp 1 or 0 when they are pressed or released. Although a contact
may appear to close firmly and quickly; at the fast running speed of a microprocessor, the action is comparatively slow. As the contact closes, the contact bounces like a ball. This bouncing effect produces multiple pulses as shown in Figure 3.1. The duration of the bounce typically will last between 5 and 10 ms. If multiple keys are needed, each switch can be connected to its own input port on the microprocessor. As the number of switches increases, however, this method quickly begins to use up all the input ports.

**Figure 3.1 Keyboard switch.**

![Keyboard switch diagram](image)

The most efficient way to lay out the switches in a keyboard (when more than five keys are needed) is to form a two-dimensional matrix as shown in Figure 3.2. The most optimum arrangement (where I/O lines are concerned) occurs when there are as many rows as columns, that is, a square matrix. A momentary contact switch (press button) is placed at the intersection of each row and column. The number of keys needed in the matrix is obviously application dependent. Each row is driven by a bit of an output port, while each column is pulled up by a resistor and fed to a bit on an input port.
Figure 3.2 **Keyboard matrix.**

Keyboard scanning is the process of having the microprocessor look at the keyboard matrix at a regular interval to see if a key has been pressed. Once the processor determines that a key has been pressed, the keyboard scanning software filters out the bounce and determines which of the keys was pressed. Each key is assigned a unique identifier called a scan code. The scan code is used by your application to determine what action is to be taken based on the key pressed. In other words, the scan code tells your application which key was pressed.

Pressing (accidentally or deliberately) more than one key at a time is called **rollover.** Any algorithm that can correctly recognize that a new key has been pressed — even though 11 keys are already pressed — is said to have **n-key rollover capability.** The matrix keyboard scan algorithms presented in this chapter do not implement an **n-key rollover algorithm** because of the extra code required. The code presented here is intended for small embedded systems where user input would occur one keystroke after the other. Such systems typically do not require full-featured keyboards like the ones found on terminals or computer systems.

### 3.01 Matrix Keyboard Scanning Algorithm

Owing initialization, all rows (output port) are forced low (see Figure 3.2). When no key is pressed, all columns (input port) read high. Any key closure will cause one of the columns to go low. To see if a key has been pressed, the microprocessor only needs to see if any of the input lines are low. Once the
microprocessor has detected that a key has been pressed, it needs to find out which key it was. This process is quite simple. The microprocessor outputs a low on only one of the rows. If it finds a 0 on the input port, the microprocessor knows that the key closure occurred on the selected row. Conversely, if the input port had all highs, the key pressed was not on that row and the microprocessor selects the next row, repeating the process until it finds the row. Once the row has been identified, the specific column of the pressed key can be established by locating the position of the single low bit on the input port. The time required for the microprocessor to perform these steps is very short compared to the minimum switch closure time and it is thus assumed that the key will remain pressed during that interval.

To filter through the bouncing problem, the microprocessor samples the keyboard at regular intervals, typically between 20 ms and 100 ms (called the debounce period) depending on the bounce characteristics of the switches being used.

The scan code of the key pressed is typically placed in a buffer until the application is ready to process a keystroke. Buffering is a handy feature because it prevents losing key presses when the application cannot process them as they occur. The size of the buffer depends on your application requirements. A buffer size of 10 keystrokes is a good starting point. The buffer is generally implemented as a circular queue. When a key is pressed, the scan code is placed at the next empty location in the queue. When your application obtains a scan code from the keyboard module, the scan code is extracted from the oldest location in the queue. If the queue is full, any further keystrokes are lost.

Another nice feature is what is called auto-repeat or typewriter. Auto-repeat allows the scan code of a key pressed to be repeatedly inserted into the buffer for as long as you press the key or until the buffer fills up. Auto-repeat capability is nice to have if you plan on incrementing or decrementing the value of a parameter (i.e., a variable) without having to continuously press and release the key. The timing diagram of Figure 3.3 shows how auto-repeat works. The scan code of the key pressed is inserted in the buffer as soon as the closure is detected. If the key is held down longer than the auto-repeat start delay, the scan code is again inserted in the buffer. From then on, if the key remains pressed, the scan code will be inserted in the buffer every auto-repeat delay.

**Figure 3.3** Auto-repeat.

Scan code placed in buffer

Key RELEASED

Auto repeat delay

Key PRESSED

By also telling you how long the key has been pressed, your application can speed up the process of incrementing or decrementing the value of a parameter based on how long the key has been pressed.

To reduce the recurring cost of your system, you can assign multiple functions to each key. To access the alternate function of each key, you can either assign a prefix key (like calculators) or provide one or more Shift keys. With a prefix key, you access the alternate function by pressing the prefix key followed by the desired key. To execute another alternate function, you generally have to press the prefix key again. With a Shift key, you access the alternate function by first pressing and holding down the Shift key and then pressing the desired key. In both cases, the keyboard scanning code can keep track of the operation and provide your application with a unique scan code for each type of key pressed. The matrix
keyboard module supports the second method and allows you to have up to three Shift keys. Note that you can still use the prefix keys with the keyboard module except that your user interface software will have to keep track of them.

## 3.02 Matrix Keyboard Module

The source code for the matrix keyboard module is found in the `\SOFTWARE\BLOCKS\KEYJAVA\SRC\` directory. The source code is found in the files `KEY.C` and `KEY.H`. The source code is shown in Listing 3.1 (KEY.C) and Listing 3.2 (KEY.H). As a convention, all functions and variables related to the keyboard module start with `key` while all `Matrix` constants start with `KEY_`.

The code allows you to scan a keyboard having any number of rows and columns up to an 8x8 matrix. Rows are driven by an output port (up to 8 bits). The module assures that rows are populated starting with bit 0 or the output port. Columns are fed to an input port (up to 8 bits). As with the rows, columns must be populated starting with bit 0. You must sacrifice column input if your application requires Shift keys. The module can accommodate up to three Shift keys. Shift keys must be populated starting with bit 7 of the input port. In other words, your first Shift key should be placed on bit 7 of the input port, the next one on bit 6, and the third on bit 5.

The module in Listing 3.1 and 3.2 has been configured and tested assuming the keyboard layout shown in Figure 3.3: a 4-row by 6-column keyboard matrix with two Shift keys. Each key in the matrix has a 4-bit code associated with it (see Figure 3.4). When the Shift key is pressed, the scan code for a key is between 0 and 23 (gcl). When the Shift1 key is pressed, the scan code for each is the number shown in Figure 3.4 plus 24. Similarly, if the Shift2 key is pressed, 48 is added to the scan codes in Figure 3.4. (See Table 3.1.)
Figure 3.4 Keyboard matrix.

Table 3.1 Scan codes for keyboard shown in Figure 3.4.

<table>
<thead>
<tr>
<th>Scan code</th>
<th>Shift key(s) pressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 25</td>
<td>None</td>
</tr>
<tr>
<td>26 - 47</td>
<td>Shift1</td>
</tr>
<tr>
<td>48 - 71</td>
<td>Shift2</td>
</tr>
<tr>
<td>72 - 95</td>
<td>Shift1 and Shift2</td>
</tr>
</tbody>
</table>

3.03 Internals

Figure 3.5 shows a low diagram of the matrix keyboard module. To use this module, all you need to do is to adapt three hardware interface functions to your environment and change the value of 17 to define constants. As shown in Figure 3.5, the code assumes the presence of a real-time kernel. The keyboard scanning module only makes use of two kernel services: semaphores and time delays. You should refer to Listings 3.1 and 3.2 for the following description. A single task, KeyboardTask1(), is responsible for scan-
ning the keyboard. KeyScanTask() is created when your application calls KeyInit(). Once created, KeyScanTask() executes every KEY_SCAN_TASK_DLY milliseconds. KEY_SCAN_TASK_DLY should be set to produce a scan rate between 10 and 30 Hz (rate in Hertz is 1000 / KEY_SCAN_TASK_DLY).

**Figure 3.5 Matrix keyboard driver flow diagram.**

The simplest method I have found to scan a keyboard and implement all the features described previously is to build a simple state machine as shown in Figure 3.6. The state machine is executed every debounce period. Only one of the four states is executed every KEY_SCAN_TASK_DLY milliseconds.
Initially, the state machine is in the `KEY_STATE_UP` state. When a key is pressed, the state of the state machine changes to `KEY_STATE_DEBOUNCE`, which will execute `KEY_SCAN_TASK_DLY` milliseconds later. Notice that the operating system's (i.e., μC/OS-II) function `OSTimeDlyNTICK()` provides a convenient way to debounce and scan the keyboard at a regular interval.

After the delay, `KeyScanTask()` executes the code in the `KEY_STATE_DEBOUNCE` state, which again checks to see if the key is pressed. The state machine returns to the `KEY_STATE_UP` state if the key is released. If the key is still pressed, however, the scan code is found by calling `KeyDecode()` and inserted in the circular buffer through `KeyBufIn()`. `KeyBufIn()` discards the scan code if the buffer is already full. `KeyBufIn()` also signals the keyboard semaphore, allowing your application to obtain the scan code of the key through `KeyCtKey()`. The state machine is then changed to the `KEY_STATE_RPT_DLY` state.

The auto-repeat function will engage if the key is pressed for more than `KEY_RPT_START_DLY` scan times. In this case, the scan code is inserted into the buffer and the state is changed to the `KEY_STATE_RPT_DLY` state. If the key is no longer pressed, the state of the state machine is changed to the `KEY_STATE_DEBOUNCE` state to debounce the released key.

After a scan period, `KeyScanTask()` executes the code in the `KEY_STATE_RPT_DLY` state, where the scan code for a pressed key will be inserted into the buffer every `KEY_RPT_DLY` scan times. As with the other states, debouncing will be required if the key is released.
3.04 Interface Functions

Figure 3.7 shows a block diagram of the matrix keyboard module. Your application knows about the keyboard module only through five functions: `KeyFlush()`, `KeyGetKey()`, `KeyGetKeyDownTime()`, `KeyHit()`, and `KeyInit()`. 

**Figure 3.7 Matrix keyboard driver block diagram.**
KeyFlush()

void KeyFlush(void);

The matrix keyboard module buffer's user keystrokes until they are consumed by your application. In some instances, it may be useful to flush the buffer and start with fresh user input. In other words, you may want to throw away previously accumulated keystrokes and start with an empty keyboard buffer. You can accomplish this by calling KeyFlush() militant.

Arguments

None

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    
    for (;;) {
     KeyFlush(); /* Clear the keyboard buffer */
    }
}
3.04 Interface Functions

Figure 3.7 shows a block diagram of the matrix keyboard module. Your application knows about the keyboard module only through five functions: KeyFlush(), KeyGetKey(), KeyGetKeyDownTime(), KeyHit() and KeyInit().

**Figure 3.7 Matrix keyboard driver block diagram.**
KeyFlush()

The matrix keyboard module buffers user keystrokes until they are consumed by your application. In some instances, it may be useful to flush the buffer and start with fresh user input. In other words, you may want to throw away previously accumulated keystrokes and start with an empty keyboard buffer. You can accomplish this by calling KeyFlush().

Arguments
None

Return Value
None

Notes/Warnings
None

Example
void Task (void *data)
{
   for (;;) {
      KeyFlush();  /* Clear the keyboard buffer */
      ...
   }
}
**KeyGetKey()**

**Syntax**  
`KeyGetKey(VOID timeout);`

`KeyGetKey()` is called by your application to obtain a scan code from the keyboard module. If a key has not been pressed, the calling task will be suspended until the user presses a key or until a user-specified timeout expires; the timeout is passed as an argument to `KeyGetKey()`. If a timeout occurs, `KeyGetKey()` returns 0x8F.

**Arguments**

to is a user-specified time out specified in 'clock ticks'. To wait for ever for a key press, specify a time-out of 0.

**Return Value**

The scan code corresponding to the key pressed or 0x8F if the specified timeout period expires. The scan code returned by `KeyGetKey()` depends on whether or not any of the Shift keys are pressed, as shown in .

**Notes/Warnings**

This function will suspend the calling task until a key is pressed.

**Example**

```c
void Task (VOID *pdata)
{

    UINT scancode;

    for (;;) {
        scancode = KeyGetKey(10); /* Wait for key to be pressed */
        /* ... up to 10 ticks */
    }
}
```

KeyGetKeyDownTime()

KeyGetKeyDownTime() returns the amount of time (in milliseconds) that a key has been pressed. This function is useful to speed up the process of incrementing or decrementing the value of a parameter based on how long a key has been pressed.

The key down time is not cleared when the pressed key is released. Instead, the key down time is reset only when the next key is pressed. In other words, you can always obtain the amount of time that the last key was pressed.

Arguments

None

Return Value

The amount of time that the current key is being pressed.

Notes/Warnings

The first edition of this book returned the time the key was pressed in number of clock ticks instead of milliseconds. You will thus have to change your code if you used the previous version of this function.

Example

```c
void Task (void *data)
{
    uint16 time;
    for (;;) {
        time = KeyGetKeyDownTime(); /* See how long last key was pressed */
    }
}
```
KeyHit()

**BOOL** KeyHit(void);

KeyHit() allows your application to determine if a key has been pressed. Unlike KeyGetKey(), KeyHit() does not suspend the caller. KeyHit() immediately returns **TRUE** if a key was pressed and **FALSE** otherwise.

**Arguments**

None

**Return Value**

**TRUE** if a key is available from the keyboard buffer.

**FALSE** if no key has been pressed.

**Notes/Warnings**

None

**Example**

```c
void Task (void *data) 
{
    DMUW scancode;

    for (;;) {
        if (KeyHit()) { /* see if a key has been pressed */
            scancode = KeyGetKey(); /* Yes, get scan code */
        }
        
    }
}
```
KeyInit()

`void KeyInit(void);`

KeyInit() is the initialization code for the module and it must be called before you invoke any of the other functions. KeyInit() is responsible for initializing internal variables used by the module, initializing the hardware ports, and creating a task that will be responsible for scanning the keyboard.

Arguments
None

Return Value
None

Notes/Warnings
None

Example
```
void main (void)
{

    KeyInit(); /* initialize the keyboard handler */

}
```

3.05 Configuration

Configuration of the matrix keyboard module code involves changing the value of 17 defines and adapting three hardware-interface functions to your environment. The defines are found in KEY.H (section: User Defined Constants) and are also found in CFG.H. The defines are fully described in KEY.H. You should typically assign a low task priority to keyboard scanning.

**WARNING:**

In the previous edition of this book, you needed to specify `KEY_SCAN_TASK_DLY` in number of ticks between execution of KeyScanTask(). Because μC/OS-II provides a more convenient function (i.e., OSTimeDlyMillISec()) to specify the task execution period in hours, minutes, seconds, and milliseconds — KEY_SCAN_TASK_DLY now specifies the scan period in milliseconds instead of ticks.
To make this module as portable as possible, access to hardware ports has been isolated into three functions: KeyInitPort (), KeySelRow (), and KeyGetCol (). The matrix keyboard module can be adapted to just about any environment as long as you write these functions as described.

KeyInitPort () is responsible for initializing the I/O ports used for the rows and columns. I tested the code using an Intel 8255A PPI (Programmable Peripheral Interface). KeyInitPort () is called by KeyInit ().

KeySelRow () is used to select rows. KeySelRow () expects a single argument that can either be KEY_ALL_ROWS (to force all rows low) or a number between 0 and 7 (to force a specific row low).

KeyGetCol () reads and returns the complement of the columns input port (a 1 indicates a key pressed).

### 3.06 How to Use the Matrix Keyboard Module

Let's suppose that your application needs a keyboard, as shown in Figure 3.8. This keyboard should look somewhat familiar except for the four function keys: F1 to F4.

Before you can use any of the keyboard module's services, you must call KeyInit ():

```c
void main(void)
{
    OSInit(); /* Initialize the O.S. (uC/OS-II) */
    .
    KeyInit(); /* Initialize the keyboard module */
    .
    OSStart(); /* Start multitasking (uC/OS-II) */
}
```
Once multitasking has started, the keyboard will be scanned at the rate defined by \texttt{KEY_SCAN_TASK_DELAY}.

At this point, your application task (typically some type of user interface) will call one of the four keyboard module services: \texttt{KeyGetKey()}, \texttt{KeyHit()}, \texttt{KeyFlash()}, or \texttt{KeyGetKeyUpTime()}.  

The following code, the user interface task calls \texttt{KeyGetKey()} by specifying a timeout of 0.  In this case, the user interface will be suspended until a key is pressed.  When a key is pressed, \texttt{KeyGetKey()} returns the scan code of the key pressed.  For example, if you pressed the \texttt{8} key, the scan code returned by \texttt{KeyGetKey()} would be 14 (see Figure 3.8).
void UserFPKeys (void *data)
{
    char key;

    data = getdata;
    for (;;) {
        key = KeyGetKey(0);    /* Wait for user input (no timeout) */
        switch (key) {
            case KEY_UP:
            case KEY_DOWN:
            case KEY_LEFT:
            case KEY_RIGHT:
                /* No action */
            case KEY_ESC:
                break;
            default:
                /* Scan codes */

        }
    }
}

You can map scan codes to anything you want by defining a lookup table:

    char UserKeyMapTbl[] = {
        /* F1 key */
        /* F2 key */
        /* F3 key */
        /* F4 key */
        /* F1 key */
        /* F2 key */
        /* F3 key */
        /* F4 key */
        /* 9 */
        /* 8 */
        /* 7 */
        /* 1 */
        /* 2 */
        /* 3 */
        /* 4 */
        /* 5 */
        /* 6 */
        /* 7 */
        /* 8 */
        /* 9 */
        /* 0 */
        /* # */
        /* * */
        /* F1 */
        /* F2 */
        /* F3 */
        /* F4 */
        /* 1 */
        /* 2 */
        /* 3 */
        /* 4 */
        /* 5 */
        /* 6 */
        /* 7 */
        /* 8 */
        /* 9 */
    }

The user interface code would now look as shown following this paragraph. With UserKeyMap tbl [ ],
the key would now be returned to your application as ASCII '9' or '9', the # would be returned as ASCII
'9', etc.
void UserIPMask (void *data)
{
    int16 code;
    char key;

    data = data;
    for (; ; ) {
        code = KeyGetKey(0); /* Wait for user input */
        key = UserKeyMapTbl[code]; /* Get ASCII value of key */
        switch (key) {
        
        
        }
    }
}

One of the disadvantages of the user interface code shown previously is that the user interface code
is suspended until a key is pressed. If your user interface also needs to display run-time information, you
can run the user interface code at a regular rate and poll the keyboard module:

void UserIPSeek (void *data)
{
    INT8U code;
    char key;

    data = data;
    for (; ; ) {
        DelaysSleepMSM(1000); /* Delay user I/F */
        if (KeyHit()) { /* See if key was pressed */
            code = KeyGetKey(0); /* Get user input */
            key = UserKeyMapTbl[code]; /* Convert to ASCII key */
            switch (key) {
            
            }
        }
        /* User interface display functions */
    }
}
3.07 Bibliography

Dybowski, John
"Negotiating a Keyboard Interface"
The Computer Applications Journal, October/November 1992, p.88-93

Lipovski, G. J.
Single- and Multiple-Chip Microcomputer Interfacing
Englewood Cliffs, NJ
Prentice Hall

Texas Instruments
TM57000 Keyboard Interface (SPN4003)
Houston, TX
Texas Instruments, 1985

Zaks, Rodney
Microprocessors, from Chips to Systems
Berkeley, CA
Sybex
Listing 3.1 KEY.C

/*
Embedded systems Building Blocks
Complete and Ready-to-use Modules in C

Nokia Keyboard Driver

(c) Copyright 1993, Sean J. LaBrousse, Newton, MA.
All Rights Reserved

Filename: KEY.C
Programmer: Sean J. LaBrousse

DESCRIPTION

The keyboard is assumed to be a matrix having 4 rows by 6 columns. However, this code works for any
matrix arrangements up to an 8 x 8 matrix. By using from one to three of the column input pins, the driver
can accept "shortcut" keys. These keys are: SHIFT, CTRL and ALT.

VAR APPLICATION software must declare (see KEY.H):

KEY_MAX_ROW  Size of the Keypad buffer
KEY_MAX_COL  Maximum number of columns on the keypad
KEY_MAX_ROWS Maximum number of columns on the keypad
KEY_MAX_REP  Number of times before auto repeat executes the function again
KEY_MAX_DELAY Number of times before auto repeat function engages
KEY_SCAN_ROW[] The number of milliseconds between keyboard scans
KEY_SCAN_TIME The size of the keyboard scanning task stack
KEY_SCAN_TIME деятельность The size of the keyboard scanning task stack

KEY_SHIFT_KEY The mask which determines which column input handles the SHIFT key
(KEY_SHIFT_KEY) The scan code offset to add when the SHIFT key is pressed
KEY_CTRL_KEY The mask which determines which column input handles the CTRL key
(KEY_CTRL_KEY) The scan code offset to add when the CTRL key is pressed
KEY_ALT_KEY The mask which determines which column input handles the ALT key
(KEY_ALT_KEY) The scan code offset to add when the ALT key is pressed

KEY_KBIO_RW The port address of the keyboard matrix ROM
KEY_KBIO_RW The port address of the keyboard matrix ROM
KEY_KBIO_RW The port address of the keyboard 1/0 port control word

KeyInit(), KeysDown() and KeysOn/Off() are the only three hardware specific functions. This has
been done to localize the interface to the hardware in only three non-features and thus make it
easier to adapt to your application.

*******************************************************************************/

/*bery*/
Listing 3.1 (continued)  KEY.C

/*
   **********************************************************************************************
   * INCLUDE FILES
   *
   */
   #include "includes.h"

   /**********************************************************************************************
   * LOCAL CONSTANTS
   *
   */
   #define KEY_WAITING 1 /* Key scan/Alg states used in KeyScan() */
   #define KEY_DOWN    2
   #define KEY_UP      3
   #define KEY_ENTER   4

   /**********************************************************************************************
   * GLOBAL VARIABES
   *
   */
   static ITMFU KeyBuf[KEY_MAX_SIZE]; /* Keyboard buffer */
   static ITMFU KeyBufEnd; /* Index into key buf where next scan code will be inserted */
   static ITMFU KeyBufSize; /* Index into key buf where next scan code will be popped */
   static ITMFU KeyDwntime; /* Count how long key has been pressed */
   static ITMFU KeyPressed; /* Number of keys read from the keyboard */
   static ITMFU KeyUpWaitTime; /* Number of scan times before key repeat is started */
   static ITMFU KeyUpRelTime; /* Number of scan times before you repeat a key once again */
   static ITMFU KeyStateStatus; /* Current state of key scanning function */
   static OLMEM KeyStateStack[KEY_MAX_STACK_SIZE]; /* Keyboard scanning stack */
   static OLMEM KeyState; /* Pointer to keyboard semaphore */

   /**********************************************************************************************
   * LOCAL FUNCTION PROTOTYPES
   *
   */
   static void KeyBufInsert(UDMFU code); /* Insert scan code into keyboard buffer */
   static INTRU KeyStateGet(); /* Get scan code from current key pressed */
   static INTRU KeyStateDown(UDMFU); /* See if key has been pressed */
   static void KeyStateTask(void *data); /* Keyboard scanning task */

*/
 Listing 3.1 (continued)  KEY.C

/*
 * INSERT KEY CHARACTER INTO KEYBOARD BUFFER
 * Description : This function inserts a key character into the keyboard buffer
 * Arguments : code  is the keyboard scan code to insert into the buffer
 * Return : none
 */

static void mysfnfill (int code)
{
    int i,
    if (sysread < KEYBUF_SIZE) {
        sysread += code;
        if (sysread < KEYBUF_SIZE) {
            sysread = 0;
        }
    }

    /* Start of critical section of code, Examine inputs */
    /* Make sure that we don't overflow the buffer */
    /* Increment the number of keys read */
    /* Store the scan code into the buffer */
    /* Adjust index to the next scan code to put in buffer */
    /* End of critical section of code */
    /* Signal user if scan code inserted in the buffer */
    /* Buffer is full, key scan code is lost */
    /* End of critical section of code */

} /*END*/
Listing 3.1 (continued)  KEY.C

/*
 ** KEY.C
 *
 ** Description: This function is called to determine the key scan code of the key pressed.
 ** Argument(s): none
 ** Return(s): the key scan code
 **
 */

static INT16 KeyScanCode (void) {
    INT8 col;
    INT8 row;
    INT8 offset;
    BOOLEAN done;
    INT8 col_off;
    INT8 row_off;

done = FALSE;

row = 0;

while (row < KEY_ROW_MAX && done) {
    row = (INT8)(row + 1);

    if (KeyScanByRow()) {
        done = TRUE;
    } else {
        /* Select next row */
        /* Find out in which row key was pressed */
        /* Select a row */
        /* See if key is pressed in this row */
        /* We are done finding the row */
    }

    col = KeyScanCol();
    offset = 0;

    if (col & KEY_SHP_PRTM) {
        offset = KEY_SHP_PRTM_OFFSET;
        /* See if SHFT1 key was also pressed */
        /* See if SHFT2 key was also pressed */
        /* An SHFT1, SHFT2 or SHFT3 key pressed */
        /* Read column */
    }

    if (col & KEY_SHP_VERT) {
        offset = KEY_SHP_VERT_OFFSET;
        /* See if SHFT3 key was also pressed */
    }

    if (col & KEY_SHP_HORZ) {
        offset = KEY_SHP_HORZ_OFFSET;
        /* See if SHFT1 key was also pressed */
    }

    mask = 0x1;
    col_off = 0;

    done = FALSE;

    while (col_off < KEY_MAX_COLS && done) {
        done = TRUE;

        if (col & mask) {
            col_off += 1;
            col -= 1;
            mask <<= 1;
        } else {
            col_off += 1;
        }
    }

    return (row * KEY_ROW_COLS + offset + col_off); /* Return scan code */
}

/*SPACE*/
Listing 3.1 (continued)  KEY.C

/*
 * Description : This function clears the keyboard buffer
 * Arguments : none
 * Returns : none
 * 
 */

void KeyFlush (void) {
  while (HasKey()) {
    KeyGetKey(); /* While there are keys in the buffer... */
    /* ... extract the next key from the buffer */
  }
}

/*
 * Description : Get a keyboard scan code from the keyboard driver.
 * Arguments : 'fd' is the file descriptor of the key device.
 * Returns : If a key is pressed, it returns the scan code of the key.
 *           If there is no key in the buffer within the specified timeout,
 *           it returns -1.
 */

int KeySetKey (int fd) {
  int buf[KEYSIZER];
  int ch, cnt = 0;

  Signal(sigexec); /* Send a signal to the keyboard device */

  if (tfgets(buf, sizeof(buf), fd)) {
    cnt = strlen(buf);
    if (cnt < 1) return (0);
    KeyFlush();
    if (KeyScanCode(buf) == KEYCODE_SIZE) {
      cnt = 0;
    }
    KeyScanCode(buf):
  }

  return (cnt > 0 ? cnt : -1);
}

/* End of KEY */
Listing 3.1 (continued)  KEY.C

/*
 **************************************************************
 */
/* Description : This function returns the amount of time the key has been pressed. */
/* Arguments  : none */
/* Returns   : key down time in 'millisecs' */
**************************************************************
*/

pdev KayGetKeyDownTime (void)
{
  int i;

  if (KeyCount1) return (Key * KEY_JOWM, TRUE, J, J);
}

/*
 **************************************************************
 */
/* Description : This function checks to see if a key was pressed */
/* Arguments  : none */
/* Returns   : TRUE  if a key has been pressed */
/* Returns   : FALSE if no key pressed */
**************************************************************
*/

BOOL KayHit (void)
{
  int i;

  if (KeyCount1) return (decls (KeyHit) > @) ? TRUE : FALSE;
  return (TJ, T J);
}
Listing 3.1 (continued)  KEY.C

/*
 * KEYSBOARD INITIALIZATION
 */

/* Description: Keyboard initialization function. KeyInit() must be called before calling any other
 * of the user accessible functions.
 * Arguments: none
 * RETURNS: none
 */

void KeyInit (void)
{
    KeyEnable(0x2F_0000); /* Select all rows */
    KeyConditions = KEY_PAN_ON.UP; /* Keyboard should not have a key pressed */
    KeyScaled = 0; /* Clear the number of keys read */
    KeyDownThr = 0;
    KeyUpThr = 0; /* Key codes inserted at the beginning of the buffer */
    KeyOutThr = 0; /* Key codes removed from the beginning of the buffer */
    KeySwThr = 0; /* Initialize the keyboard semaphore */
    KeyStatus = 0; /* Initialize I/O ports used in keyboard driver */
    InitSerialPort(); /* Initialize serial port used in keyboard driver */
    InitScreen(); /* Initialize screen */
    InitKeyboardCheck(); /* Initialize keyboard check */
    InitKeyboard(); /* Initialize keyboard */
    KeyCount = 0; /* Initialize key count */
    KeyCountMax = 0; /* Initialize key count max */
}

/*
 * keysPress
 */

/* Description: This function checks to see if a key is pressed
 * Arguments: none
 * Returns: TRUE if a key is pressed
 * SIDE EFFECT: if a key is pressed,
 * Note: (0 <= KEY_MAX_KEYZ) - 1 is used a mask to isolate the column inputs i.e. mask off
 * the KEYPRT key.
 */

static BOOLEAN KeyIsKeyDown (void)
{
    if (KeyMap[KeyIndex] & (1 << KEY_MAX_KEYZ - 1)) { /* Key not pressed if 0 */
        KeyIndex++; /* Update key down counter */
        KeyMap[KeyIndex] &= KEY_MAX_KEYZ - 1; /* Get next key index */
        return (FALSE);
    } else { /* Side effect */
        return (FALSE);
    }
}

/*
Listing 3.1 (continued) KEY.C

/***********************************************************/
/* KEYBOARD SCANNING TASK */
/***********************************************************/
/* Description: This function contains the body of the keyboard scanning task. The task should be */
/* assigned a low priority. The scanning period is determined by KEY_SCAN_PERIOD_SEC. */
/* Arguments: "data" is a pointer to data passed to task when task is created (KEY use).
/* Returns: "keyCode" never returns. */
/* Notes: */
/* An auto repeat of the key pressed will be executed after the key has been pressed for */
/* more than KEY_REPEAT_DELAY msec times. Once the auto repeat has started, the key will */
/* be repeated every KEY_REPEAT_DELAY msec times (as long as the key is pressed). For example, */
/* if the scanning of the keyboard occurs every 50 msec and KEY_REPEAT_DELAY is set to 50 */
/* and KEY_REPEAT is set to 3, then the auto repeat function will engage after 3 seconds */
/* and will repeat every 100 msec (33 times per second). */
/***********************************************************/

/***************************************************************************/
static void KeyboardTask(void *data)
{
   ...
}
Listing 3.1 (continued)  
KEY.C

for (;;) {  /* Avoid compiler warning gcc 3.3 req. */
  OPTION() = (0, 0, 0, KEY_SCAN_TIMEOUT);  /* Delay between keyboard scans */
  switch (KeyScanState) {
    case KEY_STATE_IDLE:  /* If key is pressed */
      if (KeyScanActivated) {  /* See if need to look for a key pressed */
        KeyScanState = KEY_STATE_PRESS;  /* If key is pressed */
        KeyScanTime = 0;  /* Reset key down timer */
      }
      break;

    case KEY_STATE_PRESS:  /* Key pressed, get scan code and buffer */
      InputScanCode();  /* Get scan code */
      KeyScanState = KEY_STATE_REPEAT;  /* Send scan code in buffer */
      KeyScanDelay = KEY_SCAN_DELAY;  /* Send delay in auto-repeat function */
      KeyScan = 1;  /* Key was not pressed after all */
      break;

    case KEY_STATE_REPEAT:  /* If key is still pressed */
      if (KeyScanActivated) {  /* See if key is still pressed */
        KeyScanTime++;  /* Increment decrement counter to start of next dot */
        if (KeyScanTime == 0) {  /* If delay to auto repeat is completed */
          KeyScanState = KEY_STATE_IDLE;  /* Send delay in auto-repeat function */
          KeyScanDelay = KEY_SCAN_DELAY;  /* Load time before next repeat */
          KeyScan = 1;  /* Key was not pressed after all */
        }
      }
      break;

    case KEY_STATE_IDLE:  /* If key is still pressed */
      if (KeyScanTime > 0) {  /* See if we need to delay before next key repeat */
        KeyScanTime--;  /* Decrease the key scan code */
        InputScanCode();  /* Input scan code in buffer */
        KeyScanState = KEY_STATE_IDLE;  /* Send delay in auto-repeat function */
      }
      break;

    case KEY_STATE_KEYCODE:  /* If key is not pressed after all */
      KeyScanState = KEY_STATE_IDLE;
  }
} /*END*/
Listing 3.1 (continued)  KEY.C

* Description: This function is called to read the column port.
* Arguments: none
* Returns: the complement of the column port value, 0 if no keys pressed
***************************************************************************/

/* KEY.C

DEPOS KeyPortCol (void)

(expm &input_PORT_COL)); /* Complement column (000 indicates key is pressed) */

INITIALIZE I/O PORTS
***************************************************************************/

/* KEY.C

void KeyPortRow (void)

(output_PORT_ROW, row); /* Initializeבקש: A-OUT, B-OUT, C-OUT, D-OUT */

SELECT A ROW
***************************************************************************/

/* KEY.C

void SelectRow (unsigned row)

if (row == KEY_ROW,row) {
    output_PORT_ROW, row); /* Force all rows LOW */
    else {
        output_PORT_ROW, row); /* Force desired row LOW */
    }

Return
Listing 3.2  KEY.H

/*
 * embedded systems building blocks
 * complete and ready-to-use modules in C
 * matrix keyboard driver
 * (c) copyright 1999, jean j. lapersonne, Weston, FL
 * All Rights Reserved
 * file name: KEY.H
 * programmer: jean j. lapersonne
 */

/* USER DEFINED CONSTANTS */

/* Note: These Ahedefs would normally reside in your application specific code. */

#define CPUH 10 /* Size of the keyboard buffer */
#define KEY_PORT_X 0x0110 /* The port address of the keyboard matrix rows */
#define KEY_PORT_Y 0x0111 /* The port address of the keyboard matrix columns */
#define KEY_PORT_CW 0x0113 /* The port address of the I/O port control word */
#define KEY_MAX_ROWS 4 /* The maximum number of rows on the keyboard */
#define KEY_MAX_COLS 6 /* The maximum number of columns on the keyboard */
#define KEY_PORT_CLK 2 /* Number of scan times before auto repeat executes again */
#define KEY_GPIO_SCAN_TIME 10 /* Number of scan times before auto repeat function engages */
#define KEY_BASE_CLOCK_HZ 50 /* Number of milliseconds between keyboard scans */
#define KEY_BASE_CLOCK_HZ 50 /* Set priority of keyboard scan task */
#define KEY_BASE_CLOCK_HZ 1024 /* Size of keyboard scan task stack */
#define KEY_SHTFT_H 0x60 /* The SHTFT key is on bit 0 of the column input port */
#define KEY_SHTFT_L 24 /* The scan code offset to add when SHTFT is pressed */
#define KEY_SHTFT_H 0x60 /* The SHTFT key is on bit 0 of the column input port */
#define KEY_SHTFT_L 48 /* The scan code offset to add when SHTFT is pressed */
#define KEY_SHTFT_H 0x60 /* The SHTFT key is on bit 0 of the column input port */
#define KEY_SHTFT_L 0 /* The scan code offset to add when SHTFT is pressed */
#define KEY_ALL_ROWS 0x00 /* Select all row pins, i.e. all row inputs */
Chapter 3: Keyboards — 131

Listing 3.2 (continued)  KEY.H

/*
  FUNCTION DEFINITIONS
*/

void KeyFlush (void); /* Flush the keyboard buffer */

int KeyScan (void); /* Get a keystroke from driver if one is present, -1 else */

void KeyWatch (void); /* Set how long key has been pressed (in milliseconds) */

void KeySelect (void); /* See if a key has been pressed (FALSE if so, TRUE if not) */

void KeyInit (void); /* Initialize I/O ports */

void KeyDeInit (void); /* clean up */

void KeySelect (void); /* Select a KEH */
Chapter 4

Multiplexed LED Displays

A large number of embedded systems offer some form of display device to convey information to the user. The display can consist of anything from a light indicating that power is on, to a complex graphical display showing a representation of the process. Simple control systems can be equipped with complex displays while more complex systems can offer limited information to its users; there are no set rules as to how much information has to be displayed or how it has to be presented. The world of information display is becoming extremely complex, especially when you consider new technologies such as virtual reality.

In this chapter, I will take a very modest position and describe how to interface to LED (Light Emitting Diode) displays. Specifically, I provide you with a module that allows you to control up to 64 multiplexed LEDs. The LEDs can either be seven-segment digits or discrete devices. The module presented allows you to:

- Display limited ASCII characters using seven-segment digits.
- Display numbers.
- Turn ON or OFF individual (discrete) LEDs.

4.00 LED Displays

The Light Emitting Diode, or LED, is a semiconductor device that produces visible light when a current flows through it as shown in the schematic of Figure 4.1. The intensity of the LED is proportional to the current flowing through the LED. LEDs that produce either RED, YELLOW, GREEN, or BLUE light are now commonly available. The most common color for LEDs is RED, while BLUE LEDs have just been available in the past few years.
Figure 4.1  Turning ON an LED.

As shown in Figure 4.2, a microprocessor can easily control one or more LEDs by using an output port. LEDs are turned on by writing a 0 to the appropriate bit position of the port. Here, I assume that the port can sink the current required for each LED.

Figure 4.2  Controlling LEDs with a microprocessor.

Numbers can be displayed by using what are called seven-segment LED displays as shown in Figure 4.3. Two types of seven-segment LED displays are available: common anode and common cathode. Figure 4.2 shows a common anode arrangement, while Figure 4.3 shows a common cathode arrangement.
Figure 4.3 Common cathode seven-segment LED display.

Controlling LEDs using output ports becomes expensive when the number of digits in a display increases. Fortunately, LEDs can be multiplexed. Multiplexing simply consists of connecting the LEDs in a matrix as shown in Figure 4.4 and sending the information for each digit in succession. Each digit must be updated very quickly to give the impression that all digits are turned on at the same time. Flickering will occur if the digit update rate is too low. Updating all digits at a rate of about 60 to 100 times per second will produce good results. Multiplexing is not restricted to seven-segment LED displays. The matrix shown in Figure 4.4 also includes discrete LEDs which can be used to display status information.

For example, if the display is used in an automobile, the status LEDs can indicate whether the number being displayed represents engine RPM, vehicle speed, odometer reading, trip odometer, etc. Because of the high refresh rate needed to avoid flickering, multiplexing consumes a fair amount of CPU time.
If you need additional seven-segment digits or discrete LEDs, you can add one or more 8-bit ports. The additional port(s) can be used to control more DIGITS or SEGMENTS. Adding DIGIT ports will increase the CPU overhead but will not increase the current consumption of your system. Similarly, you can add SEGMENT ports if you prefer to reduce the overhead on the CPU. In this case, however, you will be increasing the current consumption. The software presented in this chapter can be easily adapted for either situation.

If the LED display matrix needs to be located some appreciable distance from the microprocessor, you might consider using a hardware approach. In this case, a hardware solution might be less expensive, especially if you consider the cost of the connectors and cables needed to bring the control signals to the display. The Maxim 7219 should be considered in this case. The Maxim 7219 is outlined by Jeff Bachiochi in the article, "Seven-Segment LEDs Live ON" (see "Bibliography" on page 148). Using of the Maxim 7219 would eliminate the need for a multiplexing ISR (thus reducing the CPU overhead) but the segment manipulation functions would still be applicable.

### 4.01 Multiplexed LED Display Module

The source code for the multiplexed LED display module code is found in the `\SOFTWARE\BLOCKS\LED\SOURCE` directory. The source code is found in three files: LED.C (Listing 4.1), LED.H (Listing 4.2), and LED.TA.ASM (Listing 4.3). As a convention, all functions and variables related to the display module start with `Disp` while all #defines constants start with `DISP_`.

The code allows you to multiplex up to 64 LEDs (using two 8-bit output ports). The LEDs can be either be seven-segment displays, discrete LEDs, or any combination of both. The module can easily be changed if you need to add more seven-segment digits or discrete LEDs.
4.02 Internals

The software provided does not require the presence of a real-time kernel. LED_ISR.ASM, however, increments the global variable OSInterStatC and calls OSInterExit(). OSInterExit() is used to notify µC/OS-II that an ISR has started and OSInterExit() is used to notify µC/OS-II that the ISR has completed. If you are not planning on using µC/OS-II in your application, you may delete these two lines.

Implementing multiplexing in software is fairly straightforward, as shown in Figure 4.5. Here, I assume you have less than eight digits (including status indicators). You will need a hardware timer that will generate interrupts at a rate of about

\[
\text{DISP_R_DI} \times 60 \text{ (Hz)}
\]

**Figure 4.5** LED multiplexing (block diagram).

The table DispSegTbl[] contains the segment pattern for the corresponding digit (a zero indicates that the segment will be turned on). The first entry in DispSegTbl[] contains the segment patterns for the leftmost digit. DispSegTbl3A is an index into the segment table that will point to the next digit to be displayed. DispDigitSel is a mask used to select the next digit to be displayed. Note that only one of the digits can be selected at any given time. The pseudocode for the ISR is:
void DispMax1SR (void)
{
    // Save CPU registers;
    Save CPU registers;
    // Clear timer interrupt source;
    Clear timer interrupt source;
    // Turn OFF the segments of the current digit;
    Turn OFF the segments of the current digit;
    Select the next digit to display;
    // Output the segments pattern for the digit to display;
    Output the segments pattern for the digit to display;
    // Restore CPU registers;
    Restore CPU registers;
    // Return from interrupt;
    Return from interrupt;
}

You should implement DispMax1SR() in assembly language to reduce CPU utilization. I tested a C version of DispMax1SR() on an Intel 80386 running at 16 MHz. DispMax1SR() was using up 7 percent of the processor’s time. Imagine how much time the C version of DispMax1SR() would use on an 8-bit CPU!

DispMax1SR() turns OFF the segments of the current digit before selecting the next digit. This very important step is taken to prevent what is called ghosting. If the segments were not turned OFF before the next digit is selected, the segments of the previous digit would appear briefly on the newly selected digit. DispMax1SR() is only concerned with updating the display at the desired refresh rate. How the segment patterns go into DisplayYb() is the responsibility of task-level code, specifically, the application interface functions.

Conversion of decimal or hexadecimal numbers to seven-segment patterns is very straightforward when using a lookup table, as shown in Figure 4.6. The number to convert is used as an index into Disp1CharToSegYb(). Note that a limited number of alphabetical characters can also be displayed using seven-segments. DispASCIItoSegYb(), shown in Listing 4.1, provides an ASCII to seven-segment conversion table. Note that the table starts with ASCII 0 (i.e., 0x00) and ends with ASCII 9 (0x39). To obtain the seven-segment pattern of an ASCII character, you must index the table after subtracting 0x20 from the desired ASCII character as follows:

    seg = DispASCIItoSegYb[c - 0x20];
### Figure 4.6  Hexadecimal to seven-segments lookup table.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Seven-Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>11000000</td>
</tr>
<tr>
<td>2</td>
<td>11010100</td>
</tr>
<tr>
<td>3</td>
<td>11110100</td>
</tr>
<tr>
<td>4</td>
<td>11100110</td>
</tr>
<tr>
<td>5</td>
<td>11111110</td>
</tr>
<tr>
<td>6</td>
<td>11100000</td>
</tr>
<tr>
<td>7</td>
<td>11111110</td>
</tr>
<tr>
<td>8</td>
<td>11110110</td>
</tr>
<tr>
<td>9</td>
<td>11100110</td>
</tr>
<tr>
<td>A</td>
<td>10000110</td>
</tr>
<tr>
<td>B</td>
<td>10101100</td>
</tr>
<tr>
<td>C</td>
<td>10111100</td>
</tr>
<tr>
<td>D</td>
<td>10000010</td>
</tr>
<tr>
<td>E</td>
<td>10111010</td>
</tr>
<tr>
<td>F</td>
<td>10000110</td>
</tr>
</tbody>
</table>

The ASCII to seven-segments table is very useful when you combine it with standard library functions such as `itos()`, `itos()`, `printf()`, etc. For example, you can easily display numbers (converted to ASCII with `itos()`) using the function `DispStr()` as:

```
```
void DispStr (char *s, INPU dig)
{
    INPU stat;

    while (*s && dig < DISP_N_SS ) {
        Disable Interrupts;
        stat = DispSegTbl[dig] & Ox01;
        DispSegTbl[dig++] = DispASCIItoSegTbl[*s++ & Ox20] | stat;
        Enable Interrupts;
    }
}

DispStr() needs to set the seven-segment pattern without changing the state of the status bit (i.e., bit 0) because a DispSegTbl[] entry contains both the pattern for a seven-segment digit and a status bit. This is why I mask off the upper seven bits in order to isolate the state of the status. The bit pattern for the ASCII character is then merged with the status information (ORed). Interrupts are disabled when a DispSegTbl[] entry is changed because DispSegTbl[] is a critical section. DISP_N_SS defines the number of seven-segment digits in the display. Seven-segment display patterns are also assumed to be in DispSegTbl[0] through DispSegTbl[DISP_N_SS - 1].

4.03 Interface Functions

Figure 4.7 shows a block diagram of the multiplexed LED display module. Your application interfaces to the module through five functions: DispInit(), DispClrScr(), DispStr(), DispStatSet(), and DispStatClr().

Figure 4.7 LED multiplexing driver block diagram.
Chapter 4: Multiplexed LED Display — 141

DispClrScr()

void DispClrScr(void);

DispClrScr() is called by your application to clear (i.e., turn off) the display. In other words, 
DispClrScr() blanks the display.

Arguments

None

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    
    for (;;) {
        
        DispClrScr();  /* Clear everything on the display */
        
    }
}
DispInit()

void DispInit(void);

DispInit() is the initialization code for the module and must be invoked before any of the other functions. DispInit() is responsible for initializing internal variables used by the module and initializing the hardware ports.

Arguments
None

Return Value
None

Notes/Warnings
None

Example

void main (void) 
{ 

  OSInit();
  DispInit();
  
  OSStart();
}
**DispStatClr()**

```c
void DispStatClr(uint8_t dig, uint8_t seg);
```

DispStatClr() is used to turn off a single LED. This function is the complement to DispStatSet(). This function is useful when some of the LEDs are used as status indicators or decimal points for numerical data.

### Arguments

- `dig` specifies the digit that will get its segment cleared.
- `seg` specifies the specific segment to set. `seg` corresponds to the bit position in the digit as follows:
  - 0 sets segment d (bit 0)
  - 1 sets segment g (bit 1)
  - 2 sets segment f (bit 2)
  - 3 sets segment e (bit 3)
  - 4 sets segment d (bit 4)
  - 5 sets segment c (bit 5)
  - 6 sets segment b (bit 6)
  - 7 sets segment a (bit 7)

### Return Value

None

### Notes/Warnings

You can redefine status indicators and icons to make your code clearer.

### Example

```c
void Task (void *pdata)
{
    ...
    for (;;)
    {
        DispStatClr(0, WEB_TIDPounder_ICON);
        ...
    }
}
```
DispStatSet()  

void DispStatSet(int disp, int seg, void *data);

DispStatSet() is used to turn on a single LED. This function is useful when some of the LEDs are  
used as status indicators or decimal points for numerical data.

Arguments

disp specifies the digit that will get its segment set.
seg specifies the specific segment to set. seg corresponds to the bit position in the digit as follows:

- 0 sets segment dp (bit 0)
- 1 sets segment g (bit 1)
- 2 sets segment f (bit 2)
- 3 sets segment e (bit 3)
- 4 sets segment d (bit 4)
- 5 sets segment c (bit 5)
- 6 sets segment b (bit 6)
- 7 sets segment a (bit 7)

Return Value

None

Notes/Warnings

You can define status indicators and icons to make your code clearer.

Example

void Task (void *data)
{
    
    for (;;) {
        DispStatSet(5, USER_TRIP_ODOMETER_ICON);
    
    }
}
DispsStr()

void DispsStr(int dig, char *s);

DispsStr() is called to display an ASCII string. Not all ASCII characters can be displayed using a
seven-segment display. Because of this, you must be careful in the selection of messages to display.

Arguments

dig is the starting position where the ASCII string will be displayed (0 is the first 7-segment digit, 1 is
the second digit, etc.).

s is a pointer to the ASCII string. The length of the ASCII string must not exceed the number of
seven-segment digits. For example, DispsStr[12, "Hello"]; will display the string as HELLO starting
at the third seven-segments digit. Because of the limitation of seven-segments, only the 3rd character
would appear in lower case (you should display "HELLO" instead).

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    
    for (i = 0; i < 2; i++) {
        DispsStr2, "Hello");

    }

}
4.04 Configuration

Configuring the multiplexed LED display module is fairly straightforward.

1. You need to change the value of four `#defines`. These `#defines` are found and described in `LED.h` and are also found in `CPU.h`.

2. You need to adapt three hardware interface functions to your environment. To make this module as portable as possible, access to hardware ports has been encapsulated into three functions: `DispInitPort()`, `DispOutcSeg()` and `DispOutDigit()` (described in the following paragraphs).

3. You will need a hardware timer that will interrupt the CPU at the desired multiplexing rate. The interrupt should vector to `DispOutISR()` which is defined in `LED_ISR.ASM`.

`DispInitPort()` is responsible for initializing the output ports used for the segment and the digit outputs. The code assumes two 8-bit latches such as the 74HC573. Initialization thus consists only of turning off all the segment and digit outputs. I assumed a 74HC574 over an 82C55 because of the higher current drive capability of the 74HC573. `DispInitPort()` is called by `DispInit()`.

`DispOutSeg()` is used to output the segments while `DispOutDigit()` is used to select the current digit to display. Both functions are called by the multiplexing ISR handler, `DispOutHandler()`.

To reduce the ISR processing time, the multiplexing ISR code should be written entirely in assembly language and `DispOutSeg()` and `DispOutDigit()` should be integrated in the ISR. The C code is very inefficient and would not be used in an actual implementation, however, the C code is portable.

4.05 How to Use the Multiplexed LED Display Module

Let's suppose you have a four-digit LED display and four annunciator lights as shown in Figure 4.8.

**Figure 4.8 Multiplexing LEDs.**
As shown, you must call `DispInit()` before you can use any of the multiplexed LED module's services:

```c
void main (void) {
    
    DispInit();
    
}
```

Your application can use the services provided by the multiplexed LED module immediately after `DispInit()`. Display multiplexing will start as soon as you enable interrupts. Your display should be blank because `DispInit()` clears the display buffer `DispSegTable[]`. You can display the speed as follows:

```c
void UserDisplaySpeed (void) {
    char s[9];

    DispClrScr();           /* Erase what was being displayed */
    printfa ("#d", Speed);  /* Format the speed into ASCII ... */
    DispChar(0, s[1]);      /* ... and display */
    DispCharSet(4, 1);      /* Turn ON speed indicator */
}
```

Similarly, you can display the current value of the trip odometer as shown following this paragraph. Note that the trip odometer is displayed as `###.##` and, thus, we also need to turn ON the decimal point:

```c
void UserDisplayTripOdometer (void) {
    char s[5];

    DispClrScr();           /* Erase what was being displayed */
    printfa ("#6d", TripOdometer);  /* Format trip odo. to ASCII ... */
    DispChar(0, s[1]);      /* ... and display */
    DispCharSet(4, 2);      /* Turn ON trip odo. indicator */
    DispCharSet(4, 0);      /* Turn ON decimal point */
}
```
4.06 Bibliography

Artusi, Danil
"LED display drivers interface to uCs on just three I/O lines"
EDN, November 14, 1985, p259-265

Bachioci, Jeff
"Seven-Segments LEDs Live On"
The Computer Applications Journal, March 1993, p60-66

Carnell, Tom
"Smart LEDs: The Hard Way, the Soft Way, and the Right Way"

The Hewlett-Packard Applications Engineering Staff
Optoelectronics Applications Manual
Listing 4.1  LED.C

/*
  **********************************************************************************************
  * Embedded Systems Building Blocks
  * Complete and Ready-to-Use Modules in C
  * Multiplexed LED Display Driver
  *
  * Release : LED.C
  * Programme : Jean J. Labrosse
  *
  **********************************************************************************************

  DESCRIPTION

  This module provides an interface to a multiplexed 8 segments x 8 digits' LED matrix.

  To use this driver:

  1) You must define DISPLAY:

     DISP_N_DIGS  The total number of digits to display (up to 8)
     DISP_N_SEG  The total number of seven-segment digits in the display (up to 8)
     DISP_PORT_MSK  The address of the DISPLAY's output port
     DISP_PORT_MSK  The address of the DISPLAY's output port

  2) You must allocate a hardware timer which will interrupt the CPU at a rate of at least:

     DISP_N_DIGS * 60 [ms]

     The timer interrupt mask vector is displayed in LED_IA.ASM. You may write the
     code to clear the interrupt source. The interrupt source must be cleared either in int_80H() or
     in int_81H() function.

  3) Also include DispLedSeg() and DispLEDVal() for your environment.

  **********************************************************************************************
*/

"*/
Listing 4.1 (continued)  LED.C

/*
 * INCLUDE Files
 *
 */

#include "includes.h"

/**
 * LOCAL VARIABLES
 */

static int8u DisplayMask;
static int8u DisplayPattern[16][8];

/* Index into DisplayPattern for next digit to display */

/* SCRIP*/
Listing 4.1 (continued) \textit{LED.C}

```c
/*
 * const char DisplayASCIItoSeg[51] = { /* ASCII to SEGMENT conversion table */
  0x3F, /* '1', No seven-segment conversion for exclamation point */
  0x40, /* '!', Double quote */
  0x28, /* '(', Pound sign */
  0x29, /* ')', No seven-segment conversion for dollar sign */
  0x5b, /* '{', No seven-segment conversion for percent sign */
  0x60, /* '}', No seven-segment conversion for superscript */
  0x61, /* 'a', Simple quote */
  0x62, /* 'b', Same as 'a' */
  0x63, /* 'c', No seven-segment conversion for accent over */
  0x64, /* 'd', No seven-segment conversion for plus sign */
  0x65, /* 'e', No seven-segment conversion for comma */
  0x66, /* 'f', Minus sign */
  0x67, /* 'g', No seven-segment conversion for period */
  0x68, /* 'h', No seven-segment conversion for slash */
  0x69, /* 'i', '1' */
  0x6a, /* 'j' */
  0x6b, /* 'k' */
  0x6c, /* 'l' */
  0x6d, /* 'm' */
  0x6e, /* 'n' */
  0x6f, /* 'o', No seven-segment conversion for color */
  0x70, /* 'p', No seven-segment conversion for less-than sign */
  0x71, /* 'q', Equal sign */
  0x72, /* 'r', No seven-segment conversion for greater-than sign */
  0x73, /* 's', Question mark */
  0x74, /* 't', No seven-segment conversion for commercial at-sign */
  0x75, /* 'u' */
  0x76, /* 'v', Actually displayed as 'o' */
  0x77, /* 'w' */
  0x78, /* 'x' */
  0x79, /* 'y' */
  0x7a, /* 'z' */
  0x7b, /* '{' */
  0x7c, /* '}' */
  0x7d, /* '~' */
  0x7e, /* '
*/
}*/
```
":0," Actually displayed as '0' */
"":1," "; same as '1' */
":2," "; No seven-segment conversion */
":3," "; No seven-segment conversion */
":4," "; No seven-segment conversion */
":5," "; Actually displayed as '5' */
":6," "; Actually displayed as '6' */
":7," "; No seven-segment conversion */
":8," "; No seven-segment conversion */
":9," "; No seven-segment conversion */
":10," "; No seven-segment conversion */
":11," "; No seven-segment conversion */
":12," "; No seven-segment conversion */
":13," "; No seven-segment conversion */
":14," "; No seven-segment conversion */
":15," "; No seven-segment conversion */
":16," "; No seven-segment conversion */
":17," "; No seven-segment conversion */
":18," "; No seven-segment conversion */
":19," "; No seven-segment conversion */
":20," "; No seven-segment conversion for reverse quote */
":21," "; " */
":22," "; " */
":23," "; " */
":24," "; " */
":25," "; " */
":26," "; " */
":27," "; " */
":28," "; " */
":29," "; " */
":30," "; " */
":31," "; " */

"; "  /* Page */

"LED.c"
Listing 4.1 (continued)  LED.C

```c
/*
 *
 * Multiplexed LED Displays
 *
 * Model: 5600
 *
 */

const char DisplayColors[] = {
    /*@ 16 segments to LED display conversion table @*/
    /*
    *
    * a b c d e f g i d:
    *
    * 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
    */
    47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62
};
```
Listing 4.1 (continued)  LED.C

/* ........................................................................* /
/* CLEAR THE DISPLAY */
/* */
/* Description: This function is called to clear the display. */
/* Arguments: none */
/* Returns: none */
/* ........................................................................* /

void displayClear(void) {
    for (i = 0; i < LED_MAX(LED); i++) {
        OS_Segment(DISPLAY);/* Clear the screen by setting all segments */
        displayPort(i) = segmentOff;
    }
}

/* */
/* ........................................................................* /
/* DISPLAY DRIVER INITIALIZATION */
/* */
/* Description: This function initializes the display driver. */
/* Arguments: none */
/* Returns: none */
/* ........................................................................* /

void displayInit(void) {
    DisplayInit();/* Initialize I/O ports used in display driver */
    OS_PortsInit(0);
    LEDEnable = 5;
    displayPort = 0;/* Clear the display */
}

/* * */
Listing 4.1 (continued)  LED.C

void DigitalHandler (void)
{
    char segment, digit, temp; // declarations of used variables

    // Insert code to CLEAR DISPLAY HERE
    
    // Turn off segments while changing digits
    
    // Select next digit to display
    
    // Output digit's seven-segment pattern
    
    // Adjust index to next seven-segment pattern
    
    // Index into first segments pattern
    
    // 0 will select the first seven-segment digit
    
    // Select next digit

    /*SPECIAL*/
}
Listing 4.1 (continued)  LED.C

/*
 * ***************************************************************************
 * Description: This function is called to turn off a single segment on the display.
 * Arguments: dig is the position of the digit where the segment appears (0..DISP_N_DIGS-1)
 * bit is the segment bit to turn off (0..7)
 * Return: none
 * ***************************************************************************/

void DispltOffC(uint8_t dig, uint8_t bit)
{
    OS_PIXMA_CRITICAL();
    DispOffSeg(dig) = -1 << bit;
    OS_PIXMA_CRITICAL();
}

/***************************************************************************/

/*
 * Description: This function is called to turn ON a single segment on the display.
 * Arguments: dig is the position of the digit where the segment appears (0..DISP_N_DIGS-1)
 * bit is the segment bit to turn ON (0..7)
 * Return: none
 * ***************************************************************************/

void DispltOnC(uint8_t dig, uint8_t bit)
{
    OS_PIXMA_CRITICAL();
    DispSegOnSeg(dig) = -1 << bit;
    OS_PIXMA_CRITICAL();
}

/*SPADE*/
Listing 4.1 (continued)  LED.C

```c
/*
 * DESCRIPTION: This function is called to display an ASCII string on the seven-segment display.
 * Arguments: dig is the position of the first digit where the string will appear:
 * 0 for the first seven-segment digit.
 * 1 for the second seven-segment digit.
 * ...
 * D7 for the last seven-segment digit.
 * a is the ASCII string to display
 * Notes: * Not all ASCII characters can be displayed on a seven-segment display. Consult the
 * ASCII to seven-segment conversion table (see ASCIItoSevenSeg811).
 */

void DisplayString (int dig, char *a)
{
    int i;
    while (*a != NULL) {
        i = *a - '0';
        if (i < 8)  /* DIGIT_CRITICAL[] */
            i = DIGIT_CRITICAL[i];
        else
            i = -1;
        a += i; /* Cannot display */
    }
    /* cleanup */
}
```
Listing 4.1 (continued)  LED.C

#include <avr/io.h>

void initLEDs()
{
    // 1/2 PORT D initialization
    // Description: This is called by initUART() to initialize the output ports (LED) in the LED multiplexer.
    // Arguments: none
    // Returns: none
    // Notes: (DACK) 8 bit latches are used for both the segments and digits output.

    DDRD = 0x00; // Set PORT D as output
    PORTD = 0x00; // Set PORT D to all low

    // 1/2 PORT E initialization
    // Description: This function outputs the digit selector.
    // Arguments: none
    // Returns: none

    DDRB = 0x00; // Set PORT B as output
    PORTB = 0x00; // Set PORT B to all low

    // SEGMENTS output
    // Description: This function outputs seven-segment patterns.
    // Arguments: seg is the seven-segment pattern to output
    // Returns: none

    // DAC output
    // Description: This function outputs the DAC output.
    // Arguments: none
    // Returns: none

    // LED output
    // Description: This function outputs the LED outputs.
    // Arguments: none
    // Returns: none

    // DigOut0Seg (386) web
    { // Output PORT B, segment A
        PORTB = 0x01; // Set PORT B to segment A

        // Output PORT B, segment B
        PORTB = 0x02; // Set PORT B to segment B

        // Output PORT B, segment C
        PORTB = 0x04; // Set PORT B to segment C

        // Output PORT B, segment D
        PORTB = 0x08; // Set PORT B to segment D

        // Output PORT B, segment E
        PORTB = 0x10; // Set PORT B to segment E

        // Output PORT B, segment F
        PORTB = 0x20; // Set PORT B to segment F

        // Output PORT B, segment G
        PORTB = 0x40; // Set PORT B to segment G
    }

    // Output PORT B, digit 0
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 1
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 2
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 3
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 4
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 5
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 6
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

    // Output PORT B, digit 7
    { // Output PORT B, segment 0
        PORTB = 0x00; // Set PORT B to segment 0

        // Output PORT B, segment 1
        PORTB = 0x01; // Set PORT B to segment 1

        // Output PORT B, segment 2
        PORTB = 0x02; // Set PORT B to segment 2

        // Output PORT B, segment 3
        PORTB = 0x04; // Set PORT B to segment 3

        // Output PORT B, segment 4
        PORTB = 0x08; // Set PORT B to segment 4

        // Output PORT B, segment 5
        PORTB = 0x10; // Set PORT B to segment 5

        // Output PORT B, segment 6
        PORTB = 0x20; // Set PORT B to segment 6

        // Output PORT B, segment 7
        PORTB = 0x40; // Set PORT B to segment 7
    }

Listing 4.2 LED.H

/***************************************************************
* LED.H
* This is the header file for the LED display module for the
* SPI interface on the 40-pin version of the ARM processor.
* It contains the definitions of the functions and variables used
* in the display.
* Copyright: 1999, John J. Leibowiski, Wescan, Inc.
* All Rights Reserved
***************************************************************/

#ifndef LED_H
#define LED_H

#define LED_PINS   16 /* Number of pins available for LED display */
#define LED_BAUD   115200 /* Baud rate for SPI communication */
#define LED_DATA   8 /* Data pins for LED display */
#define LED_CLK    7 /* Clock pin for LED display */

#endif

/***************************************************************
* Void DisplayCharleys();
* Void DisplayUnique();
* Void DisplayRandom();
* Void DisplayMessage();
* Void DisplayCharacter(char);
* Void DisplayNumber(int, int);
* Void DisplayString(char*, int);
***************************************************************/

#endif

/***************************************************************
*  FUNCTION PROTOTYPES
*  INTERFACE SPECIFIC
***************************************************************

void DisplayCharleys();
void DisplayUnique();
void DisplayRandom();
void DisplayMessage();
void DisplayCharacter(char);
void DisplayNumber(int, int);
void DisplayString(char*, int);
Listing 4.3  LED_TA.ASM

;-------------------------------------------------------------------------------
; Embedded Systems Building Blocks
; Complete and ready-to-use Modules in C
; Multi-channel LED Display Driver
; LED Multiplexed 8x8
; Intel 8086/8088 (LARGE mode)
; (c) Copyright 1999, Jean J. LaRousse, Weston, FL
; All Rights Reserved
;
; File : LED_TA.ASM
; By  : Jean J. LaRousse
;-------------------------------------------------------------------------------

PUBLIC _DisplayRaw
EXTERN _DisplayHandlerPara
EXTERN _DisplayHandlerParaEntry
EXTERN _DisplayHandlerEntry

MODEL Large

CODE

;-------------------------------------------------------------------------------
; OUTPUT NEXT SEGMENT PATTERN TO LED DISPLAY MATRIX
; void DisplayRaw();
;-------------------------------------------------------------------------------

_DisplayRaw PROC PARA
;
; Push EAX
; push EBX
; push ESP
; inc ESP
; mov ax, DisplayHandlerParaEntry
; mov ds, ax
; mov ax, DisplayHandlerPara
; mov es, ax
; mov ax, DisplayRaw
; mov ds, ax
; push DS
; push ESP
;
; int 21h
; pop DS
; pop ESP
; pop EBX
; pop EAX
; ret
;
_End

_DisplayRaw ENDP

END
Character LCD Modules

In this chapter, I provide you with a software module that will allow you to interface with a character LCD (Liquid Crystal Display) module. This software package works with just about any character modules based on the Hitachi HD44780 Dot Matrix LCD Controller & Driver. The module allows you to:

- Control LCD modules containing up to 80 characters.
- Display ASCII characters.
- Display ASCII strings.
- Define up to eight symbols based on a 5x7 dot matrix.
- Display bar graphs.

5.00 Liquid Crystal Displays

Liquid Crystal Displays (LCDs) are a passive display technology. This means that LCDs do not emit light but instead manipulate ambient light. By manipulating this light, LCDs can display images using very little power. This characteristic has made LCDs the preferred technology whenever low power consumption is critical. An LCD is basically a reflective part. It needs ambient light to reflect back to a user’s eyes. In applications where ambient light is low or nonexistent, a light source can be placed behind the LCD. This is known as backlighting.

Backlighting can be accomplished by either using electroluminescent (EL) or LED light sources. EL backlighting is very thin and lightweight and produces a very even light source. EL backlighting for LCDs are available in a variety of colors with white being the most popular. EL backlighting consumes very little power but requires high voltages (80 to 100 Vac). EL backlighting also have a limited life of about 2,000 to 3,000 hours. LEDs are used for backlighting and are primarily used for character modules. LEDs offer a much longer life (at least 50,000 hours) and are brighter than ELs. Unfortunately, LEDs consume more power than ELs. LEDs are typically mounted in an array directly behind the display. LEDs come in a variety of colors but yellow-green LEDs are the most common.
Controlling LCDs is a little bit trickier than controlling LEDs. LCDs are almost always controlled with dedicated hardware. Figure 5.1 shows the three types of LCDs currently available:

1. Custom displays with individual segment controls (similar to LED displays). LCDs lend themselves very well to custom displays, as shown in Figure 5.1. You can design a display with just about any type of annunciator. Where software is concerned, these types of displays are similar to LED displays because each segment is controlled individually.

2. Alphanumeric or character displays. These types of displays are currently available in modules. A module contains the LCD and the drive electronics. Character displays are composed of one to four lines of 16 to 40 character blocks. Each character block consists of a 5x8 dot matrix that is used to display any ASCII character and a limited number of symbols.

3. Full graphics displays. As with character displays full graphics displays are available in modules. Graphic modules offer the greatest flexibility in formatting data on the display. They allow for text, graphics, pictures, or any combinations of these. Because character size is defined by software, graphic modules allow any language or character font. Limitations are driven by the resolution. Graphic modules are organized in rows (horizontal) and columns (vertical) of pixels. Each pixel is addressed individually, which allows any pixel to be ON or OFF. Graphics displays are available in a wide variety of configurations from 64x32 to 640x480 pixels (columns x rows). From a software point of view, interfacing with graphics displays is at least an order of magnitude more complex than interfacing with the other two types of displays. I will not be covering graphics displays in this book.

Figure 5.1 Types of LCDs.
5.01 Character LCD Modules

A character module contains the LCD and the drive electronics. Character displays are composed of one to four lines each having between 16 and 40 character blocks. Each character block consists of a 5x8 dot matrix which is used to display any ASCII character and a limited number of symbols. In this chapter, I will be providing a software interface module for character display modules. Character modules are finding their way into a large number of embedded systems such as:

- air conditioners
- radio amplifiers
- FAX machines
- laser printers
- medical equipment
- security systems
- telephones

Because of their popularity, character modules are available from an increasing number of manufacturers, including:

- Denkison Corporation
- Optrex Corp.
- Sike Instruments
- Stanley Electric

Character modules generally have at least one thing in common; they pretty much all use the Hitachi HD44780 LCD module controller. A subset of the Hitachi HD44780 data sheet can be found on the CD-ROM. The HD44780 can interface directly with any 4- or 8-bit data bus, drives very little current (less than 1 mA), is fully ASCII-compatible, can display up to 80 characters, and contains eight user-programmable 5x8 symbols. The good news is that, when software is concerned, once a display module is written, it can be used with just about any module based on the HD44780.

The hardware interface of an LCD module is quite straightforward. LCD modules can generally interface directly with most microprocessor buses either as an I/O device or a memory-mapped I/O. The HD44780 has a 500 nS (nano-second) access time. Connecting the LCD module to the microprocessor bus is economical but becomes problematic if the display is located some distance from the microprocessor bus. In this case, parallel I/O ports can be used to interface with the LCD module, as shown in Figure 5.2. Here, I used an Intel 82C55 Programmable Peripheral Interface (PPI) controller. As shown in Figure 5.2, only 11 parallel output lines are required to interface to the LCD module. Eight of the lines are used for data transfer while the other three are used as control lines for the LCD module.
Figure 5.2 Interfacing to an LCD module.

The HD44780 takes a certain amount of time to process commands or data sent to it. The Hitachi data sheet provides you with the maximum amount of time required for each type of data transfer. Because of this, the software can simply send a command or data and wait at least the amount of time specified before sending the next command or data. Note that the HD44780 itself allows the microprocessor to read a BUSY status. The BUSY status can be read by the microprocessor to determine if the HD44780 is ready to accept another command or more data. If you can, you should make use of the BUSY capability of the HD44780 because this provides you with a true indication that the HD44780 is ready to accept another command or more data. As a precaution, however, you should still provide a timeout loop to prevent hanging up the microprocessor in case of a malfunction with the interface electronics. Unless the LCD module is directly connected to the microprocessor bus, implementing read capability with parallel I/O ports is more costly. Note that the 82C55 does have a bidirectional mode but is more complex to use. This is why the circuit shown is implemented with output ports only instead of a bidirectional data port and three control lines (i.e., RS, E, and R/W).

The interface circuit is simplified by choosing to have the CPU wait between commands and data. It turns out that this scheme also makes the software easier to write. Waiting is done using a software loop.

You might be thinking that software loops should be avoided because they are not accurate. Well, in this case, accuracy is not required. All you need to do is wait at least the amount of time specified by Hitachi before sending the next command or data. A software loop also doesn’t affect responsiveness to asynchronous events since interrupts are enabled while in the loop. (Besides, how else would you wait just 40 μs with a low end processor?)

With the hardware interface shown, the LCD module appears as two write-only registers (note that the R/W line is always low). The first write register is called the data register (when RS is high) while the other write register is called the instruction register (when RS is low). The software presented in this chapter calls the instruction register the control register. Characters to display are written to the data register. The control register allows the software to control the operating mode of the module: clear the display, set the position of the cursor, turn the display ON or OFF, etc.
5.02 Character LCD Module, Internals

The source code for the LCD module is found in the \(\text{/OPTIMIZE/\text{LISBC}}\) \text{LISBC\_SOURCE} directory. The source code is found in files \text{LISBC} (Listing 5.1) and \text{LISBC\_F} (Listing 5.2). As a convention, all functions and variables related to the display module start with \text{Disp} while all \text{GR}\text{Lines} constants start with \text{GR}.

The code allows you to interface to just about any LCD module based on the Hitachi HD44780 LCD module controller. At first view, you might think that writing a software module for an LCD module is a trivial task. This is not quite the case because the HD44780 has its quirks. The HD44780 was originally designed for a 40-character by 2 lines display (40x2) and thus has internal memory to hold 80 characters. The first 40 characters are stored at memory locations \text{0x00} through \text{0x1F} (192 to 231) while the next 40 characters are stored at memory locations \text{0x20} through \text{0x37} (128 to 167) while the next 40 characters are stored at memory locations \text{0x40} through \text{0x5F} (192 to 231). Tables 5.1 through 5.4 show the memory mapping for different LCD module configurations. The addresses are shown in decimal and are actually stored at \text{0x40}. That is, address \text{0x00} actually corresponds to \text{0x40}, address \text{0x64} is actually \text{0x80} (i.e., \text{0x40} + \text{64}), etc.

Table 5.1 shows the memory organization for 16-character displays. Notice how the 16 characters by 1 line module appears as a two-line display. This is done by the LCD module manufacturers to reduce the cost of their product by fully using the drive capability of the HD44780.

| 16 Characters x 1 lines |   00  | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  |
|-------------------------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 16 Characters x 2 lines | 00   | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  |
| 16 Characters x 4 lines | 00   | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  |
| 16 Characters x 8 lines | 00   | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  |

Table 5.2 shows the memory organization for 20-character displays. Again, the single-line display appears as a two-line module.

| 20 Characters x 1 lines |   00  | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  |
|-------------------------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 20 Characters x 2 lines | 00   | 01  | 02  | 03  | 04  | 05  | 06  | 07  | 08  | 09  | 10  | 11  | 12  | 13  | 14  | 15  | 16  | 17  | 18  | 19  |

1. Memory locations inside the HD44780 chip.
### Table 5.2 20-character LCD modules

<table>
<thead>
<tr>
<th>20 Characters x 4 lines</th>
<th>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Characters x 2 lines</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</td>
</tr>
</tbody>
</table>

Table 5.3 shows the memory organization for 24-character displays. As with the 16- and 20-character displays, the single-line display appears as a two-line module.

### Table 5.3 24-character LCD modules

<table>
<thead>
<tr>
<th>24-Characters x 1 line</th>
<th>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-Characters x 2 lines</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</td>
</tr>
</tbody>
</table>

Table 5.4 shows the memory organization for 40-character displays. As with the other module configurations, the single-line display appears as a two-line module. Note that each line of a 40-character display is shown broken down into two separate lines; the second line is offset from the first. This has been done to avoid reducing the character font in order to fit within the width of the page.

### Table 5.4 40-character LCD modules

<table>
<thead>
<tr>
<th>40 Characters x 1 line</th>
<th>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 Characters x 2 lines</td>
<td>00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19</td>
</tr>
</tbody>
</table>

The software module provided in this book will support any LCD module that is organized as shown in Tables 5.1 through 5.4. The software was actually tested with an Optrex 1300C-2503A. Table 5.5 shows a list of available LCD module configurations and their manufacturer’s part numbers.
Table 5.5  LCD module configurations available.

<table>
<thead>
<tr>
<th>#Lines</th>
<th>#Characters</th>
<th>Dunixtron PN</th>
<th>Optrex PN</th>
<th>Sipek PN</th>
<th>Stanley PN</th>
<th>FEMA PN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>LM4620</td>
<td>DMC16117A</td>
<td>M1641</td>
<td>GMD1610</td>
<td>MDL1611</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>LM4622</td>
<td>DMC16207</td>
<td>M1632</td>
<td>GMD1620</td>
<td>MDL1621</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>LM4443</td>
<td>DMC16633</td>
<td>M1644</td>
<td>GMD1640</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>LM4532</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>LM4261</td>
<td>DMC20115</td>
<td>L2012</td>
<td>GMD2026</td>
<td>MDL2027</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>LM4521</td>
<td>DMC20434</td>
<td>L2014</td>
<td>GMD2040</td>
<td>MDL2041</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>LM4513</td>
<td>DMC24138</td>
<td>-</td>
<td>-</td>
<td>MDL2411</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>LM4227</td>
<td>DMC24227</td>
<td>L2432</td>
<td>GMD4220</td>
<td>MDL4221</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>LM4414</td>
<td>-</td>
<td>L4041</td>
<td>-</td>
<td>MDL4011</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>LM4518</td>
<td>DMC3028</td>
<td>L4042</td>
<td>GMD4020</td>
<td>MDL4021</td>
</tr>
</tbody>
</table>

5.03 Interface Functions

Figure 5.3 shows a block diagram of the LCD module. Your application knows about the display only through the interface functions provided.

Figure 5.3  LCD module driver block diagram.

The module assumes the presence of a real-time kernel because it requires a semaphore and time delay services. The display module makes use of a binary semaphore to prevent multiple tasks from accessing the display at the same time. Use of the semaphore is exemplified in the code, and thus, your application doesn’t have to worry about it.
**DispChar()**

```c
void DispChar(int row, int col, char c);
```

`DispChar()` allows you to display a single character anywhere on the display.

**Arguments**

- `row` and `col` will specify the coordinates (row, col) where the character will appear. `rows` (i.e., lines) are numbered from 0 to `DispMaxRows - 1`, and columns are numbered from 0 to `DispMaxCols - 1`.
- `c` is the character to display. The Hitachi H344780 allows you to specify up to eight characters or symbols numbered from 0 to 7 (i.e., its identification). You display a user-defined character or symbol by calling `DispChar()`, the row/column position, and the character or symbol's identification number.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *data) {
    ...
    for (;;) {
        DispChar(1, 3, 'G'); /* Display 'G' on second row, 4th character */
        ...
    }
}
```
DispClrLine() allows your application to clear one of the LCD module's lines. The line is basically filled with the ASCII character ' ' (i.e., \x20).

**Arguments**

线 is the line (i.e., row) to clear. Note that lines are numbered from 0 to DispMaxRows - 1.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *pdata) {
    for (;;) {
        DispClrLine(0); /* Clear the first line of the display */
    }
```
DispClrScr() allows you to clear the screen. The cursor is positioned on the top leftmost character. The screen is basically filled with the ASCII character ‘ ‘ (i.e., 0x20).

Arguments
None

Return Value
None

Notes/Warnings
None

Example

void Task (void *pdata)
{
    
    for (;;)
    {
        DispClrScr();  /* Clear everything on the display */
        
        
    }
}
**DispDefChar()**

```c
void DispDefChar(uint8_t id, uint8_t *pat);  
```

`DispDefChar()` allows you to define up to eight custom 5×8 pixel characters or symbols. This is one of the most powerful features of the LCD modules because it allows you to create graphics such as icons, bar graphs, arrows, etc.

Figure 5.4 shows how to define a character or a symbol. The 5×8 pixel matrix is organized as a bitmap table. The first entry of the table corresponds to pixels for the first row, the second entry, the pixels for the second row, etc. A pixel is turned ON when its corresponding bit is set (i.e., 1).

**Figure 5.4 Defining characters, or symbols.**

<table>
<thead>
<tr>
<th>B7 B6 B5 B4 B3 B2 B1 B0</th>
<th>Bit Map Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>[0]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[1]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[2]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[3]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[4]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[5]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[6]</td>
</tr>
<tr>
<td>0 0 0</td>
<td>[7]</td>
</tr>
</tbody>
</table>

All you need to do to define a new character or symbol is to declare an initialized array of INTRU8 containing eight entries and call `DispDefChar()`.

**Arguments**

- `id` specifies an identification number for the new character or symbol (a number between 0 and 7). The identification number will be used to actually display the new character or symbol.
- `pat` is a pointer to the bitmap table which defines what the character or symbol will look like.

**Return Value**

None

**Notes/Warnings**

None
Example

```c
const DWORD DispRightArrowChar[] = {
    0x68, 0x6C, 0x6E, 0x6F, 0x6F, 0x6E, 0x6C, 0x68
};

void Task (void *pdata)
{
    . .
    for (; ; ) {
        dispOutChar(0, dispRightArrowChar[0]); /* Define arrow char. */
    . .
    }
}
```

Figure 5.5 shows examples of bitmap to create arrows and other symbols. Once symbols are created, you can display them by calling `dispChar()`. 

**Figure 5.5  Symbol examples.**
Chapter 5: Character LCD Modules — 173

`DispHorBar()`

You can use the LCD module to create remarkably high quality bargraphs. The linear bargraph is an excellent trend indicator and can greatly enhance operator feedback. Depending on the size of the module, many bargraphs can be simultaneously displayed. The LCD module software allows you to display bargraphs of any size anywhere on the screen.

`DispHorBar()` is used to display horizontal bars anywhere on the screen.

Figure 5.6 also shows that a 16xN-character display can produce bargraphs with up to 80 bars (16 x 5 bars per character block). In Figure 5.6, I started the bargraph on the first column on a 16xN-character display. Once scaled, bargraphs can represent just about anything. For example, the 38 bars shown in Figure 5.6 can represent 47.5 percent (38 bars = 47.5% / 80), 100.7 degrees if the bargraph is used to represent temperatures from 0 to 212 degrees, etc.

**Figure 5.6  Bargraphs with 16-character displays.**

Arguments

`row` and `col` will specify the coordinates (`row`, `col`) where the first character in the bargraph will appear. Rows (i.e., lines) are numbered from 0 to `DispMaxRows - 1` and columns are numbered from 0 to `DispMaxCols - 1`.

`val` is the number of bars you want to have turned on (a number between 0 to 80 in this example).

Return Value

None
Notes/Warnings
Before you can use DispHorBar(), you must call DispHorBarInit() which defines 5 characters used for bar graphs.

Example
You could actually use fewer bars and display the actual value next to the bargraph, as shown in Figure 5.7. In this example, I am displaying 190.7 degrees (28 bars) on a scale of 0 to 212 degrees (60 bars).

**Figure 5.7  Bargraph with value.**

```c
void Task (void *data)
{
    for (i; i < 60; i++)
        DispHorBar(0, 4, 28); /* Display a 28 out of 60 bar bargraph */
}
```
DisphorBarInit()

void DisphorBarInit(void);

DisphorBarInit() defines five special symbols with identification numbers 1 through 5 as shown in Figure 5.6. You must be call before you use DisphorBar(). You only need to call DisphorBarInit() once, unless you intend to redefine the symbol identifiers dynamically for other purposes.

Arguments
None

Return Value
None

Notes/Warning
Because DisphorBarInit() defines the five symbols shown in Figure 5.6, you must use other identification numbers (i.e., 0, 6, and 7) for your own symbols.

Example

void Task (void *data)
{
    DisphorBarInit(); /* Initialize the bargraph capability */
    for (i;) {
        DisphorBar(0, 4, 28); /* Display a 28 out of 60 bar bargraph */
    }
DispInit()

void DispInit(int rows, int cols);

DispInit() is the initialization code for the module and must be invoked before any of the other functions. DispInit() assumes that multitasking has started because it uses services provided by the real-time kernel.

DispInit() initializes the hardware, creates the semaphore, and sets the operating mode of the LCD module.

Arguments

rows is the LCD module’s maximum number of rows (lines), and cols is the maximum number of columns (characters per line).

Return Value

None

Notes/Warnings

None

Example

You should call DispInit() from your user interface task as follows:

    void UserTask (void *data)
    {
        DispInit(4, 20); /* initialise the 4x20 LCD display */
        for (;;) {
            
            User interface code;
            
        }
    }
dispStr()
void dispStr(int8u row, int8u col, char *str);

dispStr() allows you to display ASCII strings anywhere on the display. You can easily display either integer or floating-point numbers using the standard library functions *cos(), lcos(), sprintf(), etc. Of course, you should ensure that these functions are reentrant if you are using them in a multitasking environment.

Arguments
row and col will specify the coordinates (row, col) where the first character of the ASCII string will appear. Note that rows (i.e., lines) are numbered from 0 to DispMaxRows – 1. Similarly, columns are numbered from 0 to DispMaxCols – 1. The upper-left corner is coordinate 0, 0.

str is a pointer to the ASCII string. The displayed string will be truncated if the string is longer than the available space on the specified line.

Return Value
None

Notes/Warnings
None

Example
void User1Task(void *task)
{
    dispInit(4, 201); /* Initialize the 4x20 LCD display */
    for (;;) {
        dispStr(0, 5, "Hello World!");
        
    }
}
5.04 LCD Module Display, Configuration

Configuring the LCD display module is quite straightforward.

1. You need to change the value of three #defines. The #defines are found and described in LCD.h and also in CFG.h. DISP_DELAY_CTRL is used to adjust delays between sending commands or data to the HD44780. You will need to change this constant so that a delay of at least 40 µs occurs between writes to the HD44780.

2. You need to adapt three hardware interface functions to your environment. To make this module as portable as possible, access to hardware ports has been encapsulated into the following functions: DisplayPort(), DisplayData(), and DispSel() (described as follows).

DisplayPort() is responsible for initializing the output ports used to interface with the LCD module. Use an Intel 82C55 PPI to verify the code. DispPort() is called by DispPort().

DisplayData() is used to write a single byte to the LCD module. Dependent on the state of the RS line (see Figure 5.2), the byte will be either sent to the data (RS is 1) or control register (RS is 0).

Changing the state of the RS line is the responsibility of the function DispSel(). DispSel() is called by the LCD display module with one argument that can either be set to DISP_SEL_CMD_REG or DISP_SEL_DATA_REG.

5.05 LCD Module Manufacturers

- Densitron Corporation
  2039 HW 11
  Camden, SC 29020
  (803) 432-5008

- Hitachi America, Ltd.
  Electron Tube Division
  3850 Holcomb Bridge Rd.
  Norcross, GA 30092
  (404) 499-3000

- Optec Corp.
  23399-T Commerce Drive
  Suite B-8
  Farmington Hills, MI 48335
  (313) 471-6200

- Seiko Instruments USA, Inc.
  Electronic Components Division
  2990 West Lomitas Blvd.
  Torrance, CA 90505
  (310) 517-7829

- Stanley Electric
  2660 Barons Parkway
  Irvine, CA 92714
  (714) 222-0777
Listing 5.1  LCD.C

******************************************************************************
* Embedded Systems Building Blocks
* Complete and Ready-to-use modules in C
* LCD Display Module driver
* (C) Copyright 1999, Jean J. Labrosse, Weston, FL
* All Rights Reserved
******************************************************************************
* Filename: LCD.C
* Programmer: Jean J. Labrosse

******************************************************************************
* DESCRIPTION
* *
* This module provides an interface to an alphanumeric display module.
* *
* The current version of this driver supports any alphanumeric LCD module based on the:
* Hitachi HD44780 or compatible LCD controller.
* *
* This driver supports LCD displays having the following configuration:
* *
* 1 line x 16 characters  2 lines x 16 characters  4 lines x 16 characters
* 1 line x 20 characters  2 lines x 20 characters  4 lines x 20 characters
* 1 line x 32 characters  2 lines x 32 characters
* 1 line x 40 characters  2 lines x 40 characters
******************************************************************************
*/

/* program! */
Listing 5.1 (continued)  

LCD.C

/*
 ***************************************************************
* INCLUDE FILES
***************************************************************
*/

/*
 * include 'includes.h'
 *
 ***************************************************************
* LOCAL VARIABLES
***************************************************************
*/

#define DISP_CLI_DIS 0x01 /* Clr display: clears display and returns cursor home */
#define DISP_CLI_FCT 0x02 /* Function set: set 8 bit data length, 11 bit disp, 8x47 */
#define DISP_CLI_DATA 0x03 /* Entry mode: inc. display data address after writing */
#define DISP_CLI_DNS 0x0C /* Display ON/OFF: Display On, cursor OFF and a blank character */

/*
 * LOCAL VARIABLES
 ***************************************************************
 */

static char DispMaxCols; /* Maximum number of columns (i.e., characters per line) */
static char DispMaxRows; /* Maximum number of rows for the display */
static char _OrgBase; /* Display; */

static char DispX[1][] = [0x00, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00];
static char DispX[2][] = [0x02, 0x01, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00];
static char DispX[3][] = [0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00];
static char DispX[4][] = [0x00, 0x01, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00];
static char DispX[5][] = [0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01];

static void DispCharSet(char x, char y); /* Set character */

"
Listing 5.1 (continued)  LCD.C

*
="/*

** DESCRIPTION: This function is used to display a single character on the display device
** ARGUMENTS: row is the row position of the cursor in the LCD display
** col is the column position of the cursor in the LCD display
** ch is the character to display in the display at the current row/column position
** RETURNS: none
*
**-------------------------------------------------------------------------------------*/

void Display(char ch, int row, int col)
{
  int err;

  if (row < DisplayRows || col < DisplayCols) {
    OpenDisplay(line, 0, &err);
    if (err != 0) {
      fprintf(stderr, "error opening display!");
      return;
    }
    DisplaySetLow(row, col);
    DisplaySetHigh(line, &err);
    if (err != 0) {
      fprintf(stderr, "error setting display!");
      return;
    }
    DisplayChar(ch, &err);
    if (err != 0) {
      fprintf(stderr, "error displaying character!");
      return;
    }
    CloseDisplay(line, 0);
    return;
  }

  /* Clear line */

  if (line < DisplayRows) {
    OpenDisplay(line, 0, &err);
    if (err != 0) {
      fprintf(stderr, "error opening display!");
      return;
    }
    DisplaySetLow(row, col);
    DisplaySetHigh(line, &err);
    if (err != 0) {
      fprintf(stderr, "error setting display!");
      return;
    }
    DisplayChar(ch, &err);
    if (err != 0) {
      fprintf(stderr, "error displaying character!");
      return;
    }
    CloseDisplay(line, 0);
    return;
  }

  /* Display */
}
Listing 5.1 (continued)  LCD.C

/*
 ******************************************************************************************
 * Description: This function clears the display
 * Arguments: none
 * Returns: none
 ******************************************************************************************
 */

void dispClear(void)
{
    /* Initialize display */
    HIMU err;
    
    CLEAR(Display, 0, &err); /* Obtain exclusive access to the display */
    dispSel(SEL_SEG_40H, &err); /* Select the LCD display command register */
    dispReg(0x08, &err); /* Send command to LCD display to clear the display */
    delay(3); /* Delay at least 3 ms to allow the display to clear */
    dispReg(0x00, &err); /* Release access to display */

    */}/*DispClear*/
Listing 5.1 (continued)  LCD.C

stuff the screen (internal)

* Description : This function positions the cursor into the LCD buffer.
* Arguments : "row" is the row position of the cursor in the LCD display.
* "col" can be a value from 0 to 0x7f on the display.
* Returns : none

******************************************************************************

static void DiagSet(int row, int col) { /* Select LCD display command register */
    DiagSet(0x01, 0x00);
    switch (row) {
    case 0:
        if (DiagStatus >= 3) { /* Handle special case when only one line */
            DiagSet(0x00 + col);
        } else {
            DiagSet(0x00 + col);
        }
        break;
    case 1:
        DiagSet(0x01 + col);
        break;
    case 2:
        DiagSet(0x02 + DiagStatus + col);
        break;
    case 3:
        DiagSet(0x03 + DiagStatus + col);
        break;
    }
}

/*ENDP*/
void dispChar(int x, int y, byte* pat) {
    INMR err;
    INMR I;

    // Obtain exclusive access to the display
    OSGetDev(OSDISP, 0, &err);
    // Select command register
    dispEncoder(cmdReg) = 0x33;
    // Set address of OS DISP
    dispEncoder(dataReg) = x * 128 + y;
    // Write pattern into OS DISP
    dispEncoder(pat++);
    // Release access to display
}
Listing 5.1 (continued)  LCD.C

/*
 * Description : This function doesn't do anything. It is used to act like a NOP (i.e. No Operation) to
 * waste a few CPU cycles and then act as a short delay.
 * Arguments : none
 * Returns : none
 */

void Display(Hold);  

void DISPLAY(A HORIZONTAL BAR)

/*
 * Description : This function allows you to display horizontal bars (i.e. graph) on the LCD module.
 * Arguments : row: is the row position of the cursor in the LCD Display
 *              'row' can be a value from 0 to DispHeight - 1
 *              val: is the value of the horizontal bar. This value cannot exceed
 *              DispHeight * 5
 * Returns : none
 * Notes : To use this function, you must first call DisplayStart()!
 */

void DispBar(char row, char col, char val)
{

}

full = val / 5;  /* Find out how many 'full' blocks to turn ON */
frac = val % 5;  /* Compute portion of block */

if (frac == 0 && full < DispHeight / 5) (DispPrevRow, DispPrevCol);

for(row = DispStartRow; row < DispHeight; row++)
{
    while(frac > 0)  { /* Write all 'full' blocks */
        COLONIZE(DispStartCol, full);
        frac--;  /* Decrease last counter */
        full--;  
    }
}

if (frac > 0) { /* Print out a single character */
    DispStart(DispEnd);
}

DispStart(DispEnd);  /* release access to display */
Listing 5.1 (continued)  LCD.C

/*
* INITIALIZATION MONTREAL BUS
* Description : This function is used to initialize the bus graphic capability of this module. You must
* call this function prior to calling DisplayChar().
* Arguments   : none
* Returns      : none
*******************************************************************************/
void DisplayCharBus (void)
{
    DispOutChar(0,DispOutChar3());
    DispOutChar(1,DispOutChar3());
    DispOutChar(2,DispOutChar3());
    DispOutChar(3,DispOutChar4());
    DispOutChar(4,DispOutChar3());
    DispOutChar(5,DispOutChar3());
}

/*******************************************************************************/

/*
* DISPLAY DRIVER INITIALIZATION
* Description : This function initializes the display driver.
* Arguments   : none
* specifies the number of lines on the display (1 to 4)
* specifies the number of characters per line
* Notes       :
* - Display() MUST be called only when multitasking has started. This is because
* - Display() requires time delay services from the operating system.
* - Display() only be called once during initialization.
*******************************************************************************/
void DisplayInit (uint32 macro, uint16 macroChar)
{
    DisplayPort(); /* Initialize I/O ports used in display driver */
    DispMaxChar = macroChar;
    DispMaxCol = macroChar;
    DispPort = DispCharCreate(); /* Create display access semaphore */
    /* INITIALIZE THE DISPLAY MODULE */
    DispStat(DispSel_ROM, DispSel_ROM); /* Select command register. */
    OptionSelOpen(0, 0, 50); /* delay more than 15 ms after power up (50 ms should be enough) */
    DispData(DispSel_ROM, DispSel_ROM); /* Function Dec: Dec 8 bit data length, 1/16 duty, 768 data */
    DispData(DispSel_ROM, DispSel_ROM); /* Many flag cannot be checked yet */
    DispData(DispSel_ROM, DispSel_ROM); /* The above command is sent four times */
    DispData(DispSel_ROM, DispSel_ROM); /* This is recommended by Hitachi in the HD44780 data sheet */
    DispData(DispSel_ROM, DispSel_ROM);
    DispData(DispSel_ROM, DispSel_ROM);
    /*Clap On/Off; Display On, cursor Off and no SLED character*/
    DispData(DispSel_ROM, DispSel_ROM); /* Sled mode for display data address when writing */
    DispData(DispSel_ROM, DispSel_ROM); /* Send command to LCD display to clear the display */
    DispData(DispSel_ROM, DispSel_ROM); /* Delay at least 2 ms (1/400 of one second) */
}

/*SOURCE*/
Listing 5.1 (continued)  LCD.C


/
******************************************************************************
* DISPLAY AN ASCII STRING
******************************************************************************
* Description : This function is used to display an ASCII string on a line of the LCD display
* Arguments : row - the row position of the cursor on the LCD display
*   - col - the column position of the cursor on the LCD display
*   - str - a pointer to the string to write to the display
* Returns : none
******************************************************************************
/

void Display(int row, int col, char *s)
{
    IN01;      /* Description: initialize IN01 */
    IN02;      /* Description: initialize IN02 */
    int row1, col1, row2, col2, i;
    if (row < DisplayRow && col < DisplayCol) {
        deselectCharacter(row1, col1); /* Select character at position row, col */
        deselectCharacter(row2, col2); /* Select character at position row, col */
        while (i < DisplayCol && *s) { /* Write string from str to display */
            DisplayChar(*s++);          /* Send character to LCD display */
            i++;                         /* Increase index */
        }
    }
    ClearDisplay();            /* Release access to display */

    /* IN01*/
Listing 5.1 (continued)  LCD.C

/*
 * WRITE DATA TO DISPLAY DEVICE
 *
 * Description: This function sends a single BME to the display device.
 * Arguments: 'byte' is the BYTE to send to the display device
 * Returns: none
 * Notes: You may need to adjust the value of DISP_XLC莫斯 (LCD_MO) to produce a delay between
 * actions of at least 40 us. The display I used for the tests actually required a delay of
 * 80 us if 'characters' seem to appear 'stuck' on the screen, you might want to increase
 * the value of DISP_XLC莫斯.
 */

#ifndef CPU

void dispData(float64 dispData) { float64 dly;

cpu(0x0F, disp); /* Write data to display module */
cpu(0x12, disp); /* Set X like WDD */
dispDelay(); /* Delay about 1us */
cpu(0x11, disp); /* Set X like LOM */
for (dly = dispDelay(); dly > 0; dly--) { /* Delay for at least 40us */
dispDelay();
}
}
#endif

/*
 * INITIALIZE DISPLAY DRIVER (cpu) blocks
 *
 * Description: This initializes the I/O ports used by the display driver.
 * Arguments: none
 * Returns: none
 */

#ifndef CPU_C

void dispInit(void) { cpu(0x0B, disp); /* Set to Mode 0: A are output, B are inputs, C are outputs */
}
#endif
Listing 5.1 (continued)  LCD.C

/**********************************************
 *          select COMMAND ON DATA REGISTER
 * Description: This function read a BYTE from the display-device.
 * Arguments:  some
 **********************************************/

/*

#include LCD_C

="#define CDL_C""#include LCD_C"

*/

/*

*/

#include LCD_C

e again (S802 chip):
{
    #if (avl = COMP_RL_CRC) READ
        // select the command register (a low)
        if (avl = COMP_RL_CRC)
            output = [RED, 802]; /* Select the data register (a low)
    #endif

/*

*/
Listing 5.2  LCD.H

/************************************************************
** LCD Display Driver Module
**
** Copyright 1999, Jean J. Lebrasse, Nanton, FL
** All Rights Reserved
**
** Filename : LCD.H
** Programmer : Jean J. Lebrasse
************************************************************/

#define CPU_H
#define DISP_DELAY_TIME  0 /* Number of iterations to delay for 60 of software loop */
#define DISP_PORT_CMD 0x000 /* Address of the Control Word (0x20) to control RS & R */
#define DISP_PORT_DATA 0x000 /* Port address of the DATA part of the LCD Module */

#define DISP_REG_CMD_HREG 0
#define DISP_REG_DATA_HREG 1

/************************************************************
** FUNCTION PROTOTYPES
************************************************************/

void DispChar(unsigned char, unsigned char, char cl);
void ClearLine(unsigned char line);
void ClearScreen(unsigned char); void ClearDisplay(unsigned char *pal);
void Display(char *); void Display(unsigned char, unsigned char); void Display(unsigned char, unsigned char, unsigned char); void Display(unsigned char, unsigned char, unsigned char, unsigned char);

/************************************************************
** HARDWARE SPECIFIC
************************************************************/

void DisplayData(Hardware);
void DisplayPort(Hardware);
void Display(Hardware);
Chapter 6

Time-of-Day Clock

The management of time is important in many microprocessor-based embedded systems. For instance, what would VCRs (Video Cassette Recorders) be without clock/calendars to schedule the recording of television programs?

In this chapter, I will describe how I implemented a Y2K-compliant clock/calendar module. The clock/calendar module offers the following features:

- Maintains hours, minutes, and seconds.
- Contains a calendar which keeps track of: month, day, year (including leap-years), and day-of-week.
- Allows your application to obtain a timestamp to mark the occurrence of events. A timestamp is the current date and time packed into a 32-bit integer.

6.00 Clocks/Calendars

A clock/calendar is a useful module for an embedded system. If you need a clock/calendar, you have to decide whether to implement it in hardware or software.

Clock/calendar chips are readily available and most can directly interface with microprocessors. These chips accurately maintain the time-of-day, and some chips even provide a built-in calendar. Some chips include a battery and can continue to keep track of date and time even when power is removed from the unit. Clock/calendar chips generally require a crystal, which further increases the recurring cost of your system. Clock/calendar chips are manufactured by a large number of semiconductor companies such as Microelectronics, National SemiConductors, Maxim, Dallas Semiconductor, etc. Just because you have a clock/calendar chip doesn’t mean you don’t need to write any software. Your application software will still need:

- program the clock/calendar chip with the correct date and time,
- program any alarm clock functions, and
- read the current date and time.
A software-maintained clock/calendar is the best solution when your application cannot afford the extra cost associated with a clock/calendar chip, a battery, and an extra crystal. A software-implemented clock/calendar module can offer most of the benefits of a hardware approach (except that it can't maintain date and time when power is removed). A software approach requires very little ROM, RAM, and CPU time and does not add recurring cost to your system. Also, you can easily add features, such as alarm clock functions (with many alarm setpoints), timestamps, string-formating utilities to convert date and time to ASCII, etc. Software-implemented clock/calendars are found in a number of familiar appliances such as VCRs, stereo, FAX machines, microwave ovens, etc. If the microprocessor has a low-power standby mode, the software-implemented clock/calendars can be made to maintain correct date and time when the power is removed by also including a battery to power the microprocessor.

Maintaining a clock/calendar is a trivial task for a microprocessor. The first thing you will need is a periodic time source that will interrupt the microprocessor at regular intervals. Such a time source is easy to find. AC power line frequencies (50 or 60 Hz) are generally very accurate over long periods of time. For short-term accuracy, the crystal used to clock the microprocessor is also a good candidate; however, for such an application, the crystal frequency must be divided down. If your application software runs under a real-time multitasking operating system, the OS's clock tick is a convenient periodic time source.

If we assumed that the microprocessor was interrupted every one-tenth (0.1) of a second, the software simply needs to maintain integer counters for tenths of a second, seconds, minutes, hours, day, month, and year as follows. The tenths of a second is incremented every interrupt. If the counter overflows from 9 to 0, the seconds counter is incremented. If the seconds counter overflows from 59 to 0, the minutes counter is incremented, etc. Every 24 hours, the days counter is incremented. When the months counter overflows depends on the current month and also, in the case of February, on whether the year is a leap year. The following sections describe how I implemented the software for the clock/calendar module.

6.01 Clock/Calendar Module

The source code for the clock/calendar module is found in the SOFTWARE\BLOCKS\CLK\SOURCE directory. The source code is found in the files CLK.C (Listing 6.1) and CLK.H (Listing 6.2). All clock/calendar functions and variables related to this module start with CLK, while all #define constants start with CLK_.

6.02 Internals

Figure 6.1 shows a simplified flow diagram of the clock/calendar module. I assume the presence of a real-time kernel but the code can easily be modified to work in a foreground/background environment. Basically, the clock/calendar module consists of a task which executes every second. The task is responsible for updating eight variables that are maintained by the clock/calendar module. You should not directly access these variables from your application. As you might have expected, the variables updated by the clock/calendar module are:
Chapter 6: Time-of-Day Clock — 193

Figure 6.1 Clock/Calendar flow diagram.

The variable CLKTS contains the current date and time in timestamp format (described later). The date and time counters of the clock/calendar are updated by the task (CLKTask()), which executes every second. The date and time counters are considered shared resources, and a mutual exclusion semaphore (CLKsem) must be acquired to access these counters.

CLKTask() calls CLKupdateTime() to update the hours (CLKhr), minutes (CLKmin), and seconds (CLKsec) counters. CLKupdateTime() returns TRUE when the clock rolls over from 23:59:59 to 00:00:00, indicating a new day. The Boolean result is used to determine whether the date-updating function, CLKupdateDate(), is called or not.

At the completion of a day, CLKupdateDate() is called to update the month (CLKmonth), day (CLKday), year (CLKyear), and day-of-week (CLKdow) counters. Updating the date is a little bit more complicated because we need to keep track of the number of days in the current month. The current day-of-week is obtained by calling CLKupdateDayOfWeek()). The day-of-week is a number between 0 and 6,
with 0 representing Sunday. The use of a table \( \text{C scour}() \) greatly simplifies the update of the days in a month and day-of-week counters.

On a lightly loaded system, the clock module should maintain accurate time. As explained in Chapter 2, specifically in Figure 2.27 on page 96, the clock task can still slowly lose track of time if all highly priority tasks (and interrupts) require more processing time than 1 clock tick. In other words, on a heavily loaded processor, \( \text{C XTask}() \) cannot maintain time accurately the way it is currently implemented. There are two ways to fix this problem. The first and simplest way is to make the clock module task a high priority task. This means that lower priority tasks will not be serviced while the clock task is executing. In general, you should assign the highest priorities to your most critical task and not the clock task because it requires a fair amount of processing time. The processor will maintain the time-of-day correctly as long as the clock task and all high priority tasks can execute in the time between clock ticks.

The second way to fix the problem requires the use of a counting semaphore, as shown in Figure 6.2. The number of clock ticks will be "monitored" in the semaphore and thus, the clock task will eventually catch up when the load of the processor is reduced. The clock tick ISR can signal the counting semaphore every clock tick or when a whole second has elapsed. I generally prefer to encapsulate these kind of details, and thus, I wrote a function called \( \text{C C Time}() \) that can be called by the clock tick ISR every time a tick occurs. Note that you need to change \( \text{C tick}() \), which is found in the file OS_C_CPU_A.ASM located in the \( \text{\$SOFTWARE\ux5f0-uXOS-T2\ux741\ux7cf7\ux6210\ux6635\ux7801\ux7742\ux5748} \) directory of the port you will use with \( mcUOS-II \) (see www.uXOS-T2.com for details on \( mcUOS-II \) port). To use the counting semaphore, you will need to set \( \text{C U.S} \) to 0 and modify \( \text{C tick}() \) to call \( \text{C C Time}() \). Setting \( \text{C C S S} \) to 1 tells the compiler to use \( \text{C Time0}() \).

Figure 6.2 Clock/Calendar flow diagram.

A timestamp (data type \(\text{T}0\)) packs a date and time into a 32-bit variable. You can use timestamps to mark when certain events have occurred. For example, a timestamp can be used to indicate when a temperature or pressure was exceeded. You can also implement alarm clock type functions using timestamps (described later).
The format of a timestamp is shown in Figure 6.3. Even though I provide you with the format, you should not directly manipulate timestamps in your applications. Instead, you should use the functions provided by this module or add functions to this module. This allows for the format to be changed at a later time without affecting your code. You should note that the year uses six bits in the timestamp format and can thus represent only 64 years. The timestamp year is the actual year minus 2000. In other words, a year value of 5 represents 2005.

**WARNING**

In the previous edition of this book, the timestamp was based on 1990 instead of 2000. If you need to be backwards compatible with the first edition, you can change the value of CLK_TS_BASE_YEAR back to 1990 which is found at the top of CLK.C.

**Figure 6.3** Timestamp format.

<table>
<thead>
<tr>
<th>BS4</th>
<th>BS3</th>
<th>BS2</th>
<th>BS1</th>
<th>BS0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>Month</td>
<td>Day</td>
<td>Hours</td>
<td>Minutes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.99</td>
</tr>
<tr>
<td>0.63</td>
<td>1.12</td>
<td>1.31</td>
<td>0.23</td>
<td>0.59</td>
</tr>
</tbody>
</table>

(Actual year - CLK_TS_BASE_YEAR)

The timestamp format guarantees that later dates and times have larger values. You can thus easily compare timestamps for equality, greater-than, less-than, etc. This feature allows you to design an alarm clock with as many alarm trips as needed.

**6.03 Interface Functions**

Your application knows about the clock/calendar through the interface functions shown in Figure 6.4.

**Figure 6.4** Clock/Calendar module interface functions.

- CLKinit()
- CLKGetTime()
- CLKFormatTime()
- CLKSetDate()
- CLKSetTime()
- CLKGetDay()
ClkFormatDate()

void ClkFormatDate(MFDate n, char *s);

ClkFormatDate() is also provided for display purposes. This function formats the current date into an ASCII string.

Arguments

n specifies the desired format for the date. ClkFormatDate() currently supports two date formats:
    n == 1: a condensed date DD-MM-YY
    n == 2: full date including:
            day of the week ("Sunday" .. "Saturday"),
            month ("January" .. "December"),
            day of the month (1..31) and
            year (CLK_TO_RASL_YEAR .. CLK_TO_RASL_YEAR + 63).

    The format is: "DayOfWeek Month Day, Year." For example, 1/1/2000 would be displayed as: "Saturday January 1, 2000." For maximum flexibility, I implemented this function using a switch statement. This allows you to easily add code to support your own date formats. For instance, you could display the date in other languages such as French, Spanish, German, etc.

s is a pointer to the string that will receive the formatted date. You must thus allocate sufficient space for your string. The condensed format (n == 1) requires 9 characters while the other format (n == 2) requires 30 characters (including the NULL character).

Return Value

None

Notes/Warnings

If you are using a preemptive kernel, you should consider making the clock/calendar task priority lower than the application software that will call ClkFormatTime() and ClkFormatDate(). Try to figure out what would happen if you were to format the date and time (these are two separate functions) just before midnight (i.e., 23:59:59)!
Example

```c
void Task (void *pdata)
{
    char t[20];
    
    for (;;) {
        
        clockFormatDate(t, 0);
    }
}
```
ClkFormatTime() is provided for display purposes. This function formats the current time into an ASCII string.

Arguments

n specifies the desired format for the time. ClkFormatTime() currently supports two time formats:

- n = 1: 24-hour format, 'HHMMSS'
- n = 2: 12-hour format with AM/PM indication, 'HHMMSS AM'

For maximum flexibility, I implemented this function using a switch statement. This allows you to easily add code to support your own formats.

a is a pointer to the string that will receive the formatted time. You must thus allocate sufficient space for your string. The 24-hour format requires 16 characters while the 12-hour format requires 12 characters (including the NULL character).

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata) {
    char s[20];
    
    for (j = 0) {
        
        ClkFormatTime(i, &s);
        
    }
}
Chapter 6. Time-of-Day Clock – 199

_ClkFormatTS_(

void CLKFormatTS(int n, TS ts, char *a);

_ClkFormatTS_() is provided for display purposes. This function formats a timestamp into an ASCII string.

Arguments

n specifies the desired format for the timestamp. _CLKFormatTS_() supports only one timestamp format:

- n == 1: "HH:DD--YY HH:MM:SS".
- n == 2: "YYYY--MM-DD HH:MM:SS".

The time is in 24-hour format. For maximum flexibility, I also implemented this function using a
switch statement. This allows you to easily add code to support your own timestamp formats.

ts is the timestamp value that you want formatted into an ASCII string.

a is a pointer to the string that will receive the formatted timestamp. You must allocate sufficient space
for your string. The timestamp format (n == 1) requires 18 characters (including the NULL character),
the timestamp requires 21 characters for format #2 (n == 2).

Return Value

None

Notes/Warnings

In the previous edition of this book, the timestamp was based on 1990 instead of 2000. If you need to be
backwards compatible with the first edition, you can change the value of CLK_TS_BASE_YEAR back to
1990 which is found at the top of CLK.C.

Example

void task (void *pdata)
{
    TS timestamp,
    char a[20];

    for (;;) {
        
        timestamp = ClkGetTS();
        CLKFormatTS(1, timestamp, a);
        Display(0, 0, a);
    
    }
ClkGetTS() is called by your application to obtain the current date and time in timestamp format. Recall that a timestamp is a 32-bit variable that contains the date and time in a packed form.

Arguments
None

Return Value
The current date and time in timestamp format.

Notes/Warnings
In the previous edition of this book, the timestamp was based on 1990 instead of 2000. If you need to be backwards compatible with the first edition, you can change the value of CLK_TS_BASE_YEALDR back to 1990 which is found at the top of CJK.C.

Example

void Task (void *pdata)
{
    TS Timestamp;

    for (; ;)
    {
        
        timestamp = ClkGetTS();
        
    }
}
**ClkInit()**

`void ClkInit(void)`

`ClkInit()` is the initialization code for the clock/calendar. `ClkInit()` must be called before any of the other functions provided in this module. `ClkInit()` is responsible for the initialization of the clock/calendar variables and the creation of the clock/calendar task.

If you choose to have a clock/calendar chip maintain the correct date and time when power is removed (using a battery), you can use `ClkInit()` to read the contents of the clock chip and load the corresponding clock/calendar module variables when power is applied to your unit. Note that PCs use this scheme.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void main(void)
{
    ...
    ClkInit();
    ...
}
```
202 — Embedded Systems Building Blocks—Second Edition

**ClkMakeTS()**

`TS ClkMakeTS(INTH month, INTH day, INTH year, INTH hr, INTH min, INTH sec);`

`ClkMakeTS()` is called by your application to format a date and time into a timestamp. This function is useful for comparing timestamps. You would use this function to implement an alarm clock feature.

**Arguments**

- **month** specifies the month of the year and must be a number between 1 and 12.
- **day** corresponds to the day of the month and must be a number between 1 and 31.
- **year** specifies the year. Here I assume you will specify a number between `CLK_TS_BASE_YEAR` (see `CLK` and `CLK_TS_BASE_YEAR`+13. Note that the year is limited to hold 64 years because the year is stored in the timestamp using six bits.
- **hr** specifies the hours and is entered in 24-hour format, i.e., a number between 0 and 23.
- **min** specifies the number of minutes and must be between 0 and 59.
- **sec** specifies the seconds and must also be a number between 0 and 59.

**Return Value**

The desired date and time in timestamp format.

**Notes/Warnings**

In the previous edition of this book, the timestamp was based on 1990 instead of 2000. If you need to be backwards compatible with the first edition, you can change the value of `CLK_TS_BASE_YEAR` back to 1990 which is found in the top of `CLK`.

**Example**

```c
void Task(void *pData)
{
    TS alarm;

    alarm = ClkMakeTS(12, 31, 1999, 23, 59, 59);
    for (;;) {
        
        if (ClkGetTS() > alarm) {
            dispstr(0, 0, "Happy New Year!");
        }
        
    }
}
```
**CkmSetDate()**

```c
void CkmSetDate(ENUM month, ENUM day, ENUM year);
```

CkmSetDate() is used to set only the calendar portion of the clock/calendar chip. If you had a clock/calendar chip, you could use this function to also set the date of the chip.

**Arguments**

- `month` specifies the month of the year and must be a number between 1 and 12.
- `day` corresponds to the day of the month and must be a number between 1 and 31.
- `year` specifies the year. Here I assumed that you will specify a number between Ckm_TMR_BASE_YEAR and Ckm_TMR_BASE_YEAR + 53.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void main(void)
{
    ...
    CkmSetDate(1, 1, 2000);
    ...
}
```
ClkSetDateTime() is used to set the clock/calendar to the desired date and time. If you had a clock/calendar chip, you could use this function to also set the date and time of the chip.

**Arguments**

- **month** specifies the month of the year and must be a number between 1 and 12.
- **day** corresponds to the day of the month and must be a number between 1 and 31.
- **year** specifies the year. Here I assumed that you will specify a number between CLK_TS_BASE_YEAR and CLK_TS_BASE_YEAR+61.
- **hr** specifies the hours and is entered in 24-hour format, i.e., a number between 0 and 23.
- **min** specifies the number of minutes and must be between 0 and 59.
- **sec** specifies the seconds and must also be a number between 0 and 59.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void main(void)
{

    //
    // ClkSetDateTime(1, 1, 2000, 21, 59, 09);
    //
}
```
ClkSetTime()

void ClkSetTime(INTHU hr, INTHU min, INTHU sec);

ClkSetTime() is used to set only the clock portion of the clock/calendar. If you had a clock/calendar chip, you could use this function to also set the time of the chip.

Arguments

hr specifies the hours and is entered in 24-hour format, i.e., a number between 0 and 23.

min specifies the number of minutes and must be between 0 and 59.

sec specifies the seconds and must also be a number between 0 and 59.

Return Value

None

Notes/Warnings

None

Example

void main(void)
{

    ClkSetTime(23, 59, 59);

}

6.04 Clock/Calendar Module, Configuration

All you need to do to use the clock/calendar module in your application is to define the value of five
#define constants (see file CLK.h and also CDT.h), call ClockInit(), and then initialize the current
date and time for the clock/calendar.

CLK_TASK_PRIO defines the priority of CLK Task() in the multitasking environment. The task pri-
orities of the clock/calendar module would typically be set relatively low (i.e., a high number under
µCOS-II) because clocks and calendars are generally not considered critical.

CLK_TICKS defines the number of "clock ticks" needed to obtain one second. I tested the code
using an IBM-PC and the tick rate was set to 200 Hz.

CLK_TASK_STACK_SIZE defines the size of the stack allocated to the clock/calendar module task. The
number of bytes allocated for the stack is given by: CLK_TASK_STACK_SIZE times sizeof(OS_STK).

WARNING:
In the previous edition of this book, CLK_TASK_STACK_SIZE specified the size of the stack for
TaskTask() in number of bytes. µCOS-II assumes the stack is specified in stack width ele-
ments.

CLK_DATE_EN is used to allow your application to save ROM space by disabling (when set to 0) the
date updating feature of the clock/calendar module.

CLK_TST_EN is used to allow your application to save ROM space by disabling (when set to 0) the
timestamp feature of the clock/calendar module. Note that you need to enable the timestamp feature when
you enable the timestamp capability.

CLK_ISR_DLY is used to indicate that the clock/calendar module will use time delays to delay the
clock task every second (when set to 1). The clock/calendar module will be expecting signals from the
timer ISR (through clksignal.Clk()) when CLK_ISR_DLY is set to 0.

6.05 Bibliography

Viscogliosi, Roberto R.
"C-mercruce and the day of the week"

Latham, Lance
Standard C Date/Time Library; Programming the World’s Calendars and Clocks
R&D Books, Lawrence, KS, 1999
ISBN 0-87930-486-0
Listing 6.1  CLK.C

/*
 * Clock/Calendar
 * 
 * (C) Copyright 1999, John J. Labrosse, Weston, FL
 * All Rights Reserved
 * 
 * File name : CLK.C
 * Programmer : John J. Labrosse
 */

/* INCLUDE FUSE */

#define CLI_CLOCKS
/* CLI.C is inferred to allocate storage for globals */
#include "includes.h"

/*
 * LOCAL CONSTANTS
 */

#define CLI_TIL_SMTP_Lang 2000 /* Time stamp start year */

/*
 * LOCAL VARIABLES
 */

static OR_USER *CLItime; /* Semaphore used to access the time of day clock */
static OR_USER *CLIseconds; /* Counting semaphore used to keep track of seconds */
static OR_MSK CLKCheck(a: (OR_MSK_PTR_PTR_SIZE));
static OR_MSK CLIRecStart; /* Counter used to keep track of system clock ticks */

/* GIVE */
Listing 6.1 (continued)  CLK.C

/***************************************************************************/
* LOCAL DEFINES
***************************************************************************/

#ifndef CLK_DATE_DEFINES
static char *CLMTMDay[] = {
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday"
};

static CLKMONTH CLKMonth[] = {
  0,    "January",
  1,    "February",
  2,    "March",
  3,    "April",
  4,    "May",
  5,    "June",
  6,    "July",
  7,    "August",
  8,    "September",
  9,    "October",
 10,   "November",
 11,   "December"
};
#endif

/***************************************************************************/
* LOCAL FUNCTION PREVIOUS
***************************************************************************/

void CLKSetWeek0(void);
static BOOLEAN CLKInDate(void);

#if CLK_DATE_DEFINES
static BOOLEAN CLKInDate0(void);
static void CLKSetDate0(void);
#endif

/*SUNDAY*/
Listing 6.1 (continued)  CLK.C

/*
 * Format current date into string.
 * description: Formats the current date into an ASCII string.
 * arguments: n is the format type.
 * 1: will format the time as "MM-DD-YY" (needs at least 8 characters)
 * 2: will format the time as "Day Month DD, YYYY" (needs at least 18 characters)
 * 3: will format the time as "YY-MM-DD" (needs at least 10 characters)
 * p is a pointer to the destination string. The destination string must be large
 * enough to hold the formatted date.
 * returns: None.
 * notes: A "switch" statement has been used to allow you to add your own date formats. For
 * example, you could display the date in French, Spanish, German, etc. by assigning
 * numbers for these types of conversions.
 * This function assumes that strang(), sprintf(), and strlen() are reentrant.
 */

#include "Clib.h"

void CLKPrintDate (MMDD n, char *p) {
    int i;
    *p = 0;
    switch (n) {
    case 1:
        strang(p, "MM-DD-YY");/* Create the template for the selected format
        p[8] = MMStr[MM/10 * 0];/* Convert D/M/D to ASCII
        p[9] = MMStr[MM % 10 * 0];
        p[10] = DDStr[DD/10 * 0];
        p[11] = DDStr[DD % 10 * 0];
        p[12] = YStr[y/100];
        p[13] = YStr[y % 100];
        break;
    case 2:
        strang(p, "Day Month DD, YYYY");/* Get the day of the week
        switch (n, strmatch("CMMON");/* Get name of month
        if (CMStr[CM/10 * 0])
            *str[0] = CMStr[CM % 10 * 0];
        else (*str[0] = 0);/* Does not have a month name
        str[1] = 0;
        str[2] = 0;
        switch (n, strmatch("CMJAN");/* Does not have a month name
        str[1] = 0;
        str[2] = 0;
        break;
Listing 6.1 (continued)  CLK.C

    case 3:
        sprintf( "YYYY-MM-DD", t);
        s(5) = year / 1000 + '0';
        s(4) = year % 1000;
        s(3) = year % 100 + '0';
        s(2) = year % 10 + '0';
      s(1) = month / 10 + '0';
      s(0) = month % 10 + '0';
      s(6) = day / 10 + '0';
      s(7) = day % 10 + '0';
        break;
    default:
        sprintf( "%d", t);
        break;
    }
    OSDevPost(CKime);
    /* Release access to clock */
    break;

    /* INITIAL */
# Listing 6.1 (continued)  
**CLK.C**

```c
/*
 * Description:  Converts the current time into an ASCII string.
 * Arguments:   0 is the format type:
 *              0  will format the time as "HH:MM:SS" 24 hour format!
 *              (up to 8 characters)
 *              1  will format the time as "HH:MM:SS AM/PM" AM/PM indicator
 *              (up to 13 characters)
 *              2  is a quitter to the destination string. The destination string
 *              must be large enough to hold the formatted time.
 * Returns:     None.
 * Notes:       A 'default' return has been used to allow you to add your own time formats.
 *              * This function assumes that string is a pointer.
 */

#include "STDARG.H"

void CLFHIIT(char *str);  

int CLFHIIT(char *str) {  
  char hr, min, sec, AM;  
  AM = 'A';  
  hr = 0;  
  min = 0;  
  sec = 0;  
  hr = toupper(hr);  
  min = toupper(min);  
  sec = toupper(sec);  
  if (AM == 'A') {  
    AM = 'P';  
    hr += 12;  
  } else {  
    AM = 'A';  
    hr -= 12;  
  }  
  hr = hr % 12;  
  if (hr == 0) hr = 12;  
  hr = hr + '0';  
  min = min % 60;  
  sec = sec % 60;  
  hr = hr + '0';  
  min = min + '0';  
  sec = sec + '0';  
  strcpy(str, hr + min + sec + AM);  
  return;  
}
```

Chapter 6: Time-of-Day Clock — 211
Listing 6.1 (continued)  CLK.C

/*
******************************************************************
*/

/* Description: This function converts a time-stamp to an ASCII string. */
/* Arguments: n: n is the desired format number: */
/* 1: "%Y-%m-%d %H:%M:%S" (needs at least 18 characters) */
/* 2: "%Y-%m-%d %H:%M" (needs at least 15 characters) */
/* ts: ts is the time-stamp value to format */
/* a: a is the destination ASCII string */
/* Returns: None */
/* Notes: The time-stamp is a 32 bit unsigned integer as follows: */
/* - Field:          - Month-    - Day-    - Hour-    - Minute-    - Second- */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 */
/*    0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 */
/* the year is based from CAK_T0_BEG_YEAR. That is, if bits 26,25 contain 0 it really */
/* means that the year is really CAK_T0_BEG_YEAR. If bits 31,30 contain 31, the year */
/* is CAK_T0_BEG_YEAR+13. */

/*******************************************************************/

#if defined YYSTYPE && defined YYSTYPE

void CLKPrintTS( YYSTYPE n, UTS ts, char *s)
{
    UINT yr;
    UINT month;
    UINT day;
    UINT hr;
    UINT min;
    UINT sec;

    yr = (UTS_TO_MONTH_YEAR + (ts >> 26)); /* Unpack time-stamp */
    month = (ts >> 22) & 0x3F;
    day = (ts >> 17) & 0x3F;
    hr = (ts >> 12) & 0x3F;
    min = (ts >> 8) & 0x3F;
    sec = (ts & 0x3F);
    printf( "AM/%04d/%02d %02d:%02d:%02d:" , yr, month, day, hr, min, sec); /* Convert to ASCII */
}

void CLKPrintTS( YYSTYPE n, UTS ts, char *s)
{
    UINT yr;
    UINT month;
    UINT day;
    UINT hr;
    UINT min;
    UINT sec;
    struct tm *t;

    yr = yr & 0x100;
    month = month & 0x100;
    day = day & 0x100;
    hr = hr & 0x100;
    min = min & 0x100;
    sec = sec & 0x100;
    break;
}

#endif
Listing 6.1 (continued)  CLK.C

case 2:
    strcpy(s, "YYYY-MM-DD HH:MM:SS"); /* Create the template for the selected format */
    yr = yr / 1000; /* Convert DATE to ASCII */
    yr = yr % 100;
    yr = yr / 100;
    mo = yr / 10;
    ta = yr % 10;
    w = week / 10;
    ta = week % 10;
    d = day / 10;
    ta = day % 10;
    h = hr / 10;
    h = hr % 10;
    m1 = min / 10;
    m1 = min % 10;
    s1 = sec / 10;
    s1 = sec % 10;

    strcat(s, " ");
    break;
}

 reconnaissance */
Listing 6.1 (continued)  CLK.C

/*
 ********************************************
 */
/*
 * Description : This function is used to return a time-stamp to your application. The format of the
 * time-stamp is shown below:
 * 
 * Fields: --------year------ --------Month-------- --------Day-------- --------Hours-------- --------Minutes-------- --------Seconds--------
 * Bits:  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 * Arguments : None.
 * Returns : None.
 */

#define CLK_DATE_YEAR (31..26)
#define CLK_DATE_MONTH (25..22)
#define CLK_DATE_DAY (21..18)
#define CLK_DATE_HOUR (17..14)
#define CLK_DATE_MINUTE (13..10)
#define CLK_DATE_SECOND (9..0)

int CLK_DATE_YMDHMS();

int main()
{
    int time;

    // CStartProgram();
    // StartProgram();
    // ExitProgram();
    // ExitProgram();
    // return (1);
    }

}*/

/*
*/
Listing 6.1 (continued) CLK.C

/*
 * START-OF-SKY Initialization
 */

/*
 * Description: This function initializes the time module. The time of day clock task will be created
 * by this function.
 * Arguments: None
 * Returns: None.
 */

void CLKInit(void)
{
    CLKmsg = OSInitEventSt(); /* Create time of day clock semaphore */
    CLKMntCht = OSInitEventSt(); /* Create counting semaphore to signal the occurrence of, etc. */
    CLMnt = 0;
    CLHr = 0;
    CLMin = 0;
    CLSec = 0;
    CLSet = 1;
    CLVenue = 1999;
}

Result:

if (CLK.Set) & (CLK.ClkSet)
    (CLMnt.Cnt) = ((CLMnt.Cnt) % 32) + 1;

/*END*/
Listing 6.1 (continued)  

```c
/*
************************************************************************************
*  CODE TO DETECT A LEAP YEAR.
************************************************************************************
*  Determines whether the year passed as an argument is a leap year.
*  
*  If year is a leap year, return TRUE.
*  If year is not a leap year, return FALSE.
************************************************************************************
*/

#if !defined(__cplusplus) || __cplusplus < 201403L
extern "C" {
#endif

/* DETECT LEAP YEAR

   Detect whether the year passed as argument is a leap year.
   @param year The year to check for leap year.
   @returns true If year is a leap year.
   false If year is not a leap year.

   */

int is_leap_year(int year)
{
    if ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0))
    return TRUE;

    return FALSE;
}

#endif

/*...*/
Listing 6.1 (continued)  

\[\text{CLK.C}\]

\[
\begin{verbatim}
/*
 **** TIMESTAMP ****

* description: This function maps a user specified date and time into a 32 bit variable called a
* time-stamp.
* arguments:  
*  month is the desired month (1..12)
*  day is the desired day (1..31)
*  year is the desired year (CLK_YEAR_MIN..CLK_YEAR_MAX)
*  hr is the desired hour (0..23)
*  min is the desired minutes (0..59)
*  sec is the desired seconds (0..59)
* returns:  
*  A time-stamp based on the arguments passed to the function.
* notes:  
*  The time stamp is formatted as follows using a 32 bit unsigned integer:
*  
*    Field         Year       Month       Day        Hour       Min        Sec
*    -----------------------  -----------------------  -----------------------  -----------------------
*    bits:            31 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
*    - The year is based on CLK_YEAR_MIN..CLK_YEAR_MAX. That is if bits 31..26 contain 5 it really
*      means that the year is really CLK_YEAR_MIN+5. If bits 31..26 contain 13, the year is
*      CLK_YEAR_MIN+13.

***************************************************************************/

/*

// (CLK_TIM & & CLK_DATE)

t_ticks( time, day, y, m, h, m, s, sec )
{
  \[\text{find}\]

  year = (y > 1600) ? (y - (1600 << 8)) : (y << 8);

  if (time < 86399)
  return (true);  

  \[endif\]

}  /* end_ticks*/

\end{verbatim}
\]
Listing 6.1 (continued)  CLK.C

******************************************************************************

* SET DATE ONLY
*
* Description : Set the date of the time-of-day clock
* Arguments : month  is the desired month [1...12]
*             day    is the desired day [1...31]
*             year  is the desired year [CLK_NS_BASE_YEAR ... CLK_NS_BASE_YEAR+50]
* Returns : None.
* Notes : It is assumed that you are specifying a correct date (i.e. there is no range checking
*         done by this function).
******************************************************************************

*/

#define CLK_NS_BASE_YEAR 1970

void CLKsetDate (int month, int day, int year)

/*

LONGCLK(ClockID, 0, &err);
CLKmonth = month;
CLKday = day;
CLKyear = year;
CLKperiod[CLKD1];

} /*ENDif */
Listing 6.1 (continued)  

```c
/**
* Description: Set the date and time of the time-of-day clock
* Arguments:  
*  month is the desired month (1..12)
*  day is the desired day   (1..31)
*  year is the desired year   (some)
*  hr is the desired hour   (0..23)
*  min is the desired minutes (0..59)
*  sec is the desired seconds (0..59)
* Returns:     None.
* Notes:      It is assumed that you are specifying a correct date and time (i.e. there is no range
*              checking done by this function).
*/

vid CLK_SET_DAY
void CLK_setDayTime(ININT month, ININT day, ININT year, ININT hr, ININT min, ININT sec)
{
    DBMU err;
    GSVenSetCLKDevID, 0, err);
    /* Gain exclusive access to time-of-day clock
    */
    CLK_dev = month;
    CLK_year = year;
    CLK_hr = hr;
    CLK_min = min;
    CLKSec = sec;
    CLKnloopDev();  
    /* Compute the day of the week (i.e. Sunday ...)
    */
    otherloop(CLKdev);
    /* Release access to time-of-day clock
    */
    enddev;  
    /* err
    */
```

Chapter 6: Time-of-Day Clock — 219
Listing 6.1 (continued)  

```c
/*  
  * Description : Set the time-of-day clock  
  * Arguments   : hr is the desired hour (0..23)  
  *               min is the desired minutes (0..59)  
  * Returns     : None  
  * Notes       : It is assumed that you are specifying a correct time (i.e. there is no range checking)  
  */  
void CLockTime (int hr, int min, int sec) {  
  CLK_SET_CRITICAL();  
  CLK = hr;  
  CLKMin = min;  
  CLKSec = sec;  
  CLK_EXIT_CRITICAL();  
}  
/*PROC*/  
  
  /* Description : This function is called by the 'clock tick' ISR on every tick. This function is then  
  * responsible for counting the number of clock ticks per second. When a second elapses,  
  * this function will signal the time-of-day clock task.  
  * Arguments   : None.  
  * Returns     : None.  
  * Notes       : THE_CLK_TICKS must be set to the number of ticks to produce 1 second.  
  *                This would typically correspond to G4 Thời мер.  
  */  
void CTimeSignalClk(void) {  
  int64_t C;  
  if (C != 0) {  
    if (C == 10000000) {  
      C = 0;  
    }  
    OSInterruptSignal();  
  }  
}  
```
Listing 6.1 (continued)  CLK.C

;/*
 * T D O - D A Y C L O C K  T A S K
 */
/* Description: This task is created by CLKInit() and is responsible for updating the time and date.
 * Arguments: None.
 * Returns: None.
 * Notes: CLK_CLK_FREQ must be set to produce 1 second delay.
 */

void CLKTask(void *data)
{
    struct tm tms
    data = data; /* Avoid compiler warning (s/38 requirement)
    for (i = 0;
        #if CLK_USE_DELAY
            delay_CLK_USE_DELAY(0, 1, 0); /* Delay for one second
        #else
            osDelayCL1CKs(0, 0); /* Wait for one second to elapses
        #endif
    if ( !CLKUpdateTime() ) -- TMS; /* Update the TIME (i.e. 3600-SS)
        #if CLK_USE_DELAY
            delay_CLK_USE_DELAY(); /* Gain exclusive access to time-of-day clock
        #endif
    if ( !CLKUpdateDate() ) -- PM; /* And date if a new day (i.e. MM-DD-VY)
        #if CLK_USE_DELAY
            delay_CLK_USE_DELAY();
        #endif
    }
    /* Reenter*/
}
Listing 6.1 (continued)  CLK.C

/******************************************************************************
 * UPDATE THE DATE
 *
 * Description : This function is called to update the date (i.e. month, day and year)
 * Arguments : None.
 * Returns : None.
 * Notes : This function updates CLDay, CLMarch, CLYear and CLMonth.
 *******************************************************************************/

#if CLK_DATE
static void CLUpdateDate(void)
{
    // CLEARTM chron:

    // Last day of the month?
    if (CLDay <= CLMonthEnd(CLMonth, CLYear)) { /* Last day of the month? */
        if (CLMonth == 2) { /* In this February? */
            if (CLYear % 400) { /* Yes, is this a leap year? */
                if (CLDay == 29) { /* Yes, last day in February? */
                    CLDay = 1;
                    if (CLMonth == 3) { /* Yes, Set to 1st day in March */
                        CLDay = 1;
                    }
                    else
                        CLMonth = CLMonth + 1;
                }
            }
            else
                CLMonth = CLMonth + 1;
        }
        else
            CLYear = CLYear + 1;
    }
    else
    {
        /* See if we have completed a month */
        if (CLMonth == 12) { /* Yes, is this december? */
            if (CLYear == 1) { /* Yes, set month to January... */
                CLMonth = 1;
                /* ...we have a new year! */
            }
            else
                CLYear = CLYear + 1;
        }
        /* No, increment the month */
        CLMonth = CLMonth + 1;
    }
    /* Compute the day of the week (i.e. Sunday ...)

}
Listing 6.1 (continued)  

```
/*
 * ***************************************************************************
 * ** Description : This function computes the day of the week (0 -- Sunday) based on the current month,
 * ** day and year.
 * ** Arguments  : None.
 * ** Returns   : None.
 * ** Notes     : This function updates the CLK.
 * ** This function is called by CLKupdateWeek() and CLKupdateDay().
 * ***************************************************************************/
*/

Ref  CLKdate
static void CLKupdateDayNW (void)
{
    unsigned dow;
	dow = CLKday + (CLKmonth*31) + (CLKmonth*ModDay);
    if (CLKmonth > 3) {
        if (CLKLeapYear (CLKyear)) {
            dow += 5;
        }
    }
	dow = (CLKyear + (CLKyear / 4));
	dow = (CLKyear / 100) + (CLKyear / 400);
	dow = dow;

    CLKday = dow;
}

/*SPRCD*/
Listing 6.1 (continued)  CLK.C

/*
 ** Description: This function is called to update the time (i.e. hours, min., sec. and seconds) 
 ** Arguments: None. 
 ** Returns: TRUE if we have completed one day. 
 ** FALSE otherwise. 
 ** Notes: This function updates CLShr, CLMin and CLSec.
 **------------------------------------------------------------------------*/

static BOOLEAN CLdateUpdateTime (void)
{ 
  BOOLEAN today;

  today = FALSE;
  if (CLSec == 59) { 
    if (CLMin == 59) { 
      if (CLHour == 23) { 
        if (CLDay == 31) { 
          today = TRUE;
        } else { 
          CLMin++;
        }
      } else { 
       今天 = TRUE;
      }
    } else { 
      CLMin++;
    } 
  } else { 
    CLSec++;
  } 
  return (today); 
}
Listing 6.2  CLK. H

/*
   ***************************************************************************/
/*
   * File name:  CLK.H
   * Programmed: John J. Lakrome
   * Copyright 1990, John J. Lakrome, Wason, MI.
   * All Rights Reserved
   */
/*
   *****************************************************************************/
/*
   */
/*
   *****************************************************************************/
/*
   CONSTANTS
   */
/*
   *****************************************************************************/

#define CLK_H
#define CLK_Clock_TICKS 0.0001
#define CLK_TASK_PRIO 50
#define CLK_TASK_BEGIN 512
#define CLK_TASK_END 1
#define CLK_PG_END 1
#define CLK_TASK_PRIO 1

#define CLK_TASK
#define CLK_TASK
#define CLK_TASK
#define CLK_TASK
#define CLK_TASK
Listing 6.2 (continued)  

```c
#define INTH DR   
/* Definition of Time Stamp */

#if CLK дней > 0  
#define INTH @   
/* Number of days in each month */
#define MNTH@        
/* Name of the month */
#define INTH @       
/* Value used to compute day of the week */
#endif

/* GLOBAL VARIABLES */

#include

#define INTH @;  
/* Counts for local TIME */

#define INTH @;  
/* Counts for local STH */

#define INTH @;  
/* Day of week (0 is Sunday) */

#define INTH @;  
/* Counts for local STH */

#define INTH @;  
/* CURRENT TIME-GEN */

#include
```
Listing 6.2 (continued)  

CLK.H

/*
 * ********************************************************************
 *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
 */

void CLKinit(void);

void CLKnormalTime(MENU n, char *ti);
void CLKdateTime(MENU hr, MENU min, MENU sec);
void CLKtimerhr(void);

#ifdef CLK_CPLLIB
void cplGetTime(MENU n, char *a);
#else
void cplGetTime(MENU n, char *a);
#endif

cplGetDate(MENU month, MENU day, MENU year, MENU hr, MENU min, MENU sec);

#endif

#ifndef

#define CLK_CPLLIB

#endif


void CLKnormalTime(MENU n, ti, char *ti);

#endif
Chapter 7

Timer Manager

Timers are useful in situations where you start an operation, wait a certain amount of time, and then stop the operation. Usually the process looks like this:

1. Start an operation (turn on or turn off an output device).
2. Start the timer.
3. When the timer expires, stop the operation (turn OFF or turn ON the output device).

You can also use timers to detect timeout conditions. For example, you turn on a motor and then start a timer. Here, you are expecting the speed of the motor (i.e., RPM) to increase. If the speed of the motor doesn’t exceed a threshold before the timer times out, then you might turn the motor off and notify an operator. In these cases, you start an operation then monitor the process to see if conditions are met before the timer expires:

1. Start an operation.
2. Start the timer.
3. Monitor for desired conditions. If conditions are met, stop the timer.
4. If timer times out, stop the operation and notify operator.

In this chapter, I will describe how I implemented a countdown timer module. The countdown timer module provides your application with as many countdown timers as your application requires (up to 250). Each countdown timer has a resolution of 0.1 second and can be programmed to expire after 99 minutes, 59 seconds, and 0.9 seconds. Each countdown timer can be individually started, stopped, set, reset, and checked. Also a user-defined function can be executed when a countdown timer expires (one for each timer).

7.00 Timer Manager Module

The source code for the timer manager module is in the \DRIVER\LIBRARY\TM2\SOURCE directory. The source code consists of two files: \TM2.C (Listing 7.1) and \TM2.H (Listing 7.2). All timer managa...
functions and variables related to this module start with Tmr while all declare constants start with TMR_.

### 7.01 Timer Manager Moduler, Internals

Figure 7.1 shows the flow diagram of the timer manager module. Here, I assume the presence of a real-time kernel. This module consists of a single task that executes every tenth of a second. The timer manager task (TmrTask()) is responsible for updating as many countdown timers as your application requires (defined by TMR_MAX_TMR in TMR.h). You can have up to 250 timers.

**Figure 7.1 Timer manager module flow diagram.**

The timer manager is designed around the TMR data structure (TMR.h) which is declared as follows:

```c
typedef struct TMR {
    BOOLEAN TmrEn;
    INT16U TmrCnt;
    INT16U TmrInit;
    void (*TmrFunct)(void *);
    void **TmrFunctArg;
} TMR;
```
timer is used to enable and disable the countdown process. Countdown occurs when timer is set to TRUE by TimerStart(). Countdown is suspended when timer is set to FALSE by TimerStop().

When the timer is enabled, TimerTask() decrements timer to zero. When timer reaches zero, countdown stops. Timer is loaded when either TimerSet() or TimerReset() is called.

The initial value of timer is stored in Timer. Timer is changed by either TimerSet() or TimerReset().

TimerTask() is a pointer to a user-defined function that TimerTask() executes whenever its corresponding Timer reaches zero. The called function is passed TimerProc() (a pointer) as an argument. Both TimerProc() and TimerReset() are set by TimerReset() (described later). You must define your timer function as follows:

```c
void timerProc(void *arg);
```

Note that TimerProc() is passed TimerProcArg when it is called. This allows you to design a single function that can be used by more than one timer. The user-defined function will be called by the timer task TimerTask() when the timer expires. The execution time of the timer task is thus increased by the execution time of all the functions that will execute when their respective timers expire. You may defer processing of the timeout to another task because the function that executes when the timer expires can signal another task through a semaphore, a mailbox, or even a message queue, as shown:

```c
void timerProc(void *arg)
{
    TaskProc(GUI_EVENT *arg);
}
```

If you are using µC/OS-II, the argument passed to the user function (in this example) is a pointer to the semaphore.

Some applications do not require the execution of a function upon timeout. In such situations, you will not have to set the pointer because its initial value is NULL. In other words, the timer manager will not execute any function when pointing to NULL.

When the timer manager task executes, it scans all entries in TimerTbl[] for enabled timers. For each timer that has been enabled, TimerTask() decrements TimerTbl[]. TimerTask() towaused 0. If the timer reaches 0, the user-defined function (if specified) is executed.

On a lightly-loaded system, the timer manager module should maintain accurate time. As I explained in Chapter 2, specifically Figure 2.27 on page 96, the timer manager task could miss clock ticks if all higher-priority tasks (and interrupts) require more processing time than one clock tick. In other words, on a heavily loaded processor, TimerTask() cannot maintain track of time accurately the way it is currently implemented. This is the same problem as with the time-of-day clock described in Chapter 5. Unlike the clock task, however, there is really only one correct way to fix this problem. You really don't want to increase the priority of the timer manager task because its processing time does not depend only on the number of times it has to manage. Instead, the execution time of the timer manager task depends on the execution time of the functions that will be executed when each timer expires. To fix this problem, you need to use a counting semaphore, as shown in Figure 7.2.
The number of clock ticks will be "memorized" in the semaphore, and thus the timer manager task will eventually catch up when the load of the processor is reduced. The clock tick ISR can signal the counting semaphore every clock tick or when 0.1 second has elapsed. I generally prefer to encapsulate the details, so I wrote a function called TmrSignalTick(), which can be called by the clock tick ISR every time a tick occurs. Note that you need to change OSTickISR(), which is found in the file OS_CPU_A.ASM located in the SOFTWARE\uCOS-III\cpp\ compiler\ SOURCES directory of the port you will use with uCOS-II (see www.uCOS-II.com for details on uCOS-II ports). To use the counting semaphore, you will need to set TMS_USRSR0 to 1 and modify OSTickISR() to call TmrSignalTick().

If you need to manage a large number of timers then you might consider changing the implementation of the module provided in this chapter to a delta list. A delta list would maintain a linked list of only the enabled timers. The list would be ordered so that the timer with the least amount of time to timeout is first. TmrTask() would decrement the first entry in the list without scanning the list because the remaining delays are relative to it. For example, if you had five enabled timers with values of 10, 14, 21, 32 then, the list would contain 10, 4, 7, 31, and 7. The total time before the first timer would expire is 10, the second is 10+4, the third is 10+4+7, the fourth is 10+4+7+11, and finally, the fifth timer would be 10+4+7+11+7. The use of a delta list is really only justified when you need many timers. One of the drawbacks of the delta list is that you need one (for a singly-linked list) or two pointers (for a doubly-linked list). You can find a more complete discussion on delta lists in the excellent book by Douglas Comer, Operating System Design: The XINU approach.
7.02 Timer Manager Module, Interface Functions

Your application software interfaces with the timer manager through interface functions as shown in Figure 7.3.

**Figure 7.3 Timer manager module interface functions.**

- TimerInit()
- TimerPost()
- TimerSetT()
- TimerSetMT()
- TimerStart()
- TimerStop()
- TimerReset()
- TimerChk()
- TimerFormat()
TmrCfgFnct()  

```c
void TmrCfgFnct(TMR n, void (*pfnc)(void*), void *arg);
```

Each timer can execute a user-defined function when it expires. In order to use this feature, you must specify the address of the function to execute when the timer expires. This is accomplished by calling TmrCfgFnct().

The execution time of the timer task is augmented by the execution time of all the functions that will execute when their respective timers expire. Some applications do not require the execution of a function upon timeout. In these situations, there is no need to call TmrCfgFnct() because the initial value of the pointer to a function for each timer is NULL. In other words, the timer manager will not execute any function where pfnc is a NULL pointer.

**Arguments**

- `n` is the timer number to set and must be a number between 0 and TMR_MAX - 1.
- `pfnc` is a pointer to the function that you would like to execute when the timer expires. You must define this function as follows:

  ```c
  void UserFnct(void *arg);
  ```

  **Note:** UserFnct() is called with the argument you specify in TmrCfgFnct(), that is, `arg`. This allows you to design a single function that can be used by more than one timer. The user-defined function will be called by the timer task TmrTask() when the timer expires.

**Return Value**

None

**Notes/Warnings**

UserFnct() is called with interrupts enabled and you should need to protect any shared objects.
Example

void main (void) {
    //
    ThcCtxPut(0, Thc0TimeoutFact, (void *)0);
    //
    ThcSetNRT(0, 1, 0, 0); /* Set timer #0 to expire in 1 minute */
    ThcStart(0); /* Start timer #0 */
    //
}

void Thc0TimeoutFact (void *exp) {
    DispStr(0, 0, "Timer #0 expired!");
}
TmrChk()

TmrChk() allows you to check the progress of the countdown timer. Basically, the function returns the time remaining (in tenths of a second) until the timer expires. The timer expired if the returned value is 0.

Arguments
- n is the timer number to start and must be a number between 0 and THR_MAX_TMR - 1.

Return Value
- The time remaining (in tenths of a second) of the desired timer.

Notes/Warnings
- This function doesn’t tell you whether the timer is running or not.

Example

```c
void Task (void)
{
    INT16U time_remaining;

    for (;;) {
    
        time_remaining = TmrChk(0); /* Get time left for timer #0 */
    
    }
}
```
The `FmtFormat()` is provided for display purposes. This function formats the time remaining of the specified timer into an ASCII string. Timers are always formatted as follows: `HH:MM:SS,T` where `HH` is the remaining minutes to timeout, `MM` is the remaining seconds, and `SS,T` is the tenths of a second.

**Arguments**

- `n` is the timer number to format into an ASCII string and must be a number between 0 and `#MUL_TIMER - 1`.
- `s` is a pointer to the string that will receive the formatted time. Your destination string must allocate at least eight characters (including the NULL character).

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void)
{
    char #[10];

    for (;;) {
        FmtFormat(0, &#); /* Get time left for timer 0 as "HH:MM:SS,T" */
    }
}
```
**TmrInit()**

`void TmrInit(void);`

*TmrInit()* is the initialization code for the timer manager module. You must call *TmrInit()* before any other functions provided by this module. *TmrInit()* is responsible for the initialization of the timer module variables and the creation of the timer manager task.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

The `#define TMR_MAX_TMR` (see section 7.03, "Timer Module, Configuration" on page 24) defines the number of timers managed by this module. All timers are disabled and in a non-configured state following initialization.

**Example**

```c
void main(void)
{
    ...
    TmrInit();
    ...
}
```
TmrReset()

void TmrReset (TMRn n);

You can restart the countdown process to its initial value (established by either TmrSet() or TmrSetN0P()) by calling TmrReset(). This is a convenient function to use if you don’t need to reprogram the timer with a new value every time you need to use the timer.

Arguments
n is the timer number to start and must be a number between 0 and TMR_MAX_TMR = 3.

Return Value
None

Notes/Warnings
None

Example

void Task (void)
{
  for (;;) {
    ...
    TmrReset (0); /* Load timer 0 */
    ...
  }
}
TmrSetMST()

```c
void TmrSetMST(uint8_t n, int8_t min, int8_t sec, int8_t tenths);
```

This function allows you to set a timer by specifying minutes, seconds, and tenths of a second.

### Arguments

- `n` is the timer number to set and must be a number between 0 and `TMR_MAX_TIMER = 1`.
- `min` is the desired number of minutes (0..99).
- `sec` is the desired number of seconds (0..59).
- `tenths` is the desired number of tenths of a second (0..9).

### Return Value

None

### Notes/Warnings

Note that changing the timer value does not enable the timer. This means that setting the timer value does not initiate countdown. Countdown is initiated by calling `TmrStart()`. If the timer is enabled, however, `TmrSetMST()` will reload the timer and countdown will start from the new value.

### Example

```c
void Task(void)
{
    for (i = 0; i < 10; i++)
    {
        TmrSetMST(0, 0, 15, 0); /* Reset timer #0 to 15 seconds */
    }
}
```
**TimerSetT()**

`void TimerSetT(int n, int tenths);`

This function allows you to set a timer in tenths of a second.

**Arguments**

`n` is the timer number to set and must be a number between 0 and `TMR_MAX_TMR` - 1.

`tenths` is the desired timeout value of the timer and is specified in tenths of a second. For example, to set a timer to 27.4 seconds, you would specify 274.

**Return Value**

None

**Notes/Warnings**

Note that changing the timer value does not enable the timer. This means that setting the timer value does not initiate countdown. Countdown is initiated by calling `TimerStart()`. If the timer is enabled, however, `TimerSetT()` will reload the timer and countdown will start from the new value.

**Example**

```c
void track (void)
{
    for (i; )
    {
        TimerSetT(0, 150);  /* Reset timer #0 to 15 seconds */
    }
}
```
TmrStart()
void TmrStart(DWORD n);

Countdown of a timer is initiated only when you call TmrStart(). You should set the countdown time prior to calling TmrStart() with either TmrSetT() or TmrSetMST().

Arguments
n is the timer number to start and must be a number between 0 and TMR_MAX_TMR - 1.

Return Value
None

Notes/Warnings
TmrStart() will resume countdown of a timer that has been suspended by TmrStop().

Example

void Task (void)
{
    for (;;) {
        TmrSetT(0, 150);  /* Reset timer 0 to 15 seconds */
        TmrStart(0);      /* Start timer 0 */
    }
}
**Timer**

*Start* (void TimerStart(int n))

Countdown of a timer can be suspended by calling *TimerStop*. You can later resume countdown by calling *TimerStart*.

**Arguments**

*n* is the timer number to start and must be a number between 0 and *NUM_MAX_TIMER* - 1.

**Return Value**

None

**Notes/Warnings**

*TimerStop* doesn't reset the timer value, it simply suspends it.

**Example**

```c
void Task (void)
{
    for (;;) {
        TimerStop(0);  /* stop (i.e. suspend) timer 0 */
    }
}
```
7.03 Timer Manager Module, Configuration

Configuration of the timer manager consists of defining the value of four \texttt{#define} constants (see file timer.h also, CFO 89).

\texttt{TM_TASK_PRIO} defines the priority of \texttt{TaskTask() in the multitasking environment. The task priority of the timer manager module would typically be set relatively low.}

\texttt{TM_CLK_TICKS} defines the number of clock ticks needed to obtain 0.1 second. If you use \texttt{µCOS-II}, you can set this \texttt{#define} constant to \texttt{CLK\_TICKS\_PRIO\_SRC / 10}.

\texttt{TM_TASK\_STK\_SIZE} defines the size of the stack (in bus width units) allocated to the timer manager module task. The number of bytes allocated for the stack is thus given by: \texttt{TM\_TASK\_STK\_SIZE times \texttt{sizeof (BUS\_STK)}}.

\texttt{WARNING}

In the previous edition of this book, \texttt{TM\_TASK\_STK\_SIZE} specified the size of the stack for \texttt{TaskTask()} in number of bytes. \texttt{µCOS-II} assumes the stack is specified in stack width elements.

\texttt{TM\_MAX\_TM} defines the number of timers managed by \texttt{TaskTask()}. If you use this module, you will need to have at least one timer. The timer manager can manage up to 250 timers. The limitation is strictly dictated by the amount of memory available and by the addressing capability of the target microprocessor.

\texttt{TM\_USE\_SEM} is used to indicate that the timer manager will be expecting a signal from the tick ISR (through \texttt{TaskSignalTick()}). When \texttt{TM\_USE\_SEM} is set to 0, \texttt{TaskTask()} will use the kernel’s time delay service (\texttt{TaskDelay()} for \texttt{µCOS-II}).

7.04 Bibliography

Corner, Douglas

\textit{Operating System Design, The XINU Approach}

Englewood Cliffs, New Jersey

Prentice-Hall, Inc., 1984
Listing 7.1  TMR.C

/*
 * Embedded Systems Building Blocks
 * Complete and Ready-to-Use Modules in C
 * Timer Manager
 *
 * (c) Copyright 1999, Sean J. Labrosse, Weston, FL
 * All Rights Reserved
 *
 * filename  : TMR.C
 * programmer : Sean J. Labrosse
 *
 */

/*
 * INCLUDE FILES
 */

#define TMR_C
#include "includes.h"

/*
 * LOCAL VARIABLES
 */

static _ISRNAME *TaskServicePtr;
static _ISRNAME *TaskProcessorStartAddr;
static _ISRNAME *TaskStartAddr;

/*
 * TMR страх *TaskStartAddr *data);
 */

"SINGE"
Listing 7.1 (continued)  TMR.C

/

#include "time.h"

/*

** DESCRIPTION : Set the user defined function when the timer expires.
**
** Arguments :  n  is the timer number 0..MAX_TMR-1
**             func  is a pointer to a function that will be executed when the timer expires
**             arg  is a pointer to an argument that is passed to func
**
** Returns : None

**

*/

void tmrSet (void (*func)(void *, void *, void *)) {
    TMRTCOUNT = 0;
    /* Store pointer to function into timer */
    func = func;
    /* Store user's function arguments pointer */
    arg = arg;
}

/*END*/

/*

** CHECK TIMER

** DESCRIPTION : This function checks to see if a timer has expired
** Arguments :  n  is the timer to check
**
** Returns : 0 if the timer has expired
**             1 if the timer is running

**

*/

DWORD Check (DWORD n) {
    DWORD val;
    val = 0;
    if (n < MAX_TMR) {
        val = DelayEx(n, curTick);
        curTick = curTick + 1;
    }
    return val;
}

/*END*/
Listing 7.1 (continued)TIMER.C

/**
  * TURN Timer INTO STRING
  *
  * Description: Converts a Timer into an ASCII string.
  * Arguments:
  * in is the input Timer
  * dest is a pointer to the destination string.
  * The destination string must be large
  * enough to hold the formatted Timer value which will have the following format:
  */

void TimerToStr(TIMER & in, char * dest)
{
  DINTU min;
  DINTU sec;
  DINTU mSec;
  DINTU val;

  if (in < TAKE:max) {
    \n    Get local copy of timer for conversion
    \n    val = Get[\n    \n    min = \n    sec = \n    mSec = \n    \n    y10 = \n    h11 = \n    d(0) = \n    d(1) = \n    d(2) = \n    d(3) = \n    d(4) = \n    d(5) = \n    d(6) = \n    d(7) = 
  
  /* Convert Time to ASCII */
  
  /* END*/
}*/
Listing 7.1 (continued)  

```c
/*
*******************************************************************************
* TIMES MANAGER INITIALIZATION
* Description:  This function initializes the timer manager module.
* Arguments:    None
* Returns:      None.
*******************************************************************************
*/

void TimerInit( void )
{
    DMUX enx;
    DMUX l;
    TM6 *paps;

    paps = &Timer[0];
    for ( i = 0; i < TM6_MAX_CHAN; i++ ) /* Clear and disable all timers */
    {
        paps->actsIn[i] = FALSE;
        paps->chCnt[i] = 0;
        paps->chMask[i] = 0;
        paps->chPre[i] = 0;
    }

    TimerOcr = 0;
    TimerSmtCn = 0;
    /* Create counting semaphore to signal 1/32 second */
    semWrite( &TimerSmtCn, (void *)&TimerSmtCn, TM6_EXPR_SIZE, TM6_EXPR_SIZE);
}
```

/*SEIZE*/
Listing 7.1 (continued)  TMRR.C

/*
  ***************************************************************
  */
  #define TIME

  /*
  * Description: This function advances a timer with its initial value
  * Arguments: n  is the timer to update
  * Returns:  none
  
  ***************************************************************
  */

void TMRR_Count(int n)
{
  int pm;

  if (n < TMRR_MIN
      pm = pm^74(n);
      TRM = TRM - pm^72(n);
      timer->perv = pm^71(n);  /* record the current */
      return;
    }

    /*
    
    ***************************************************************
    */
    #define COUNT

    /*
    * Description: Set the timer with the specified number of
    * minutes, seconds and 1/10 seconds.  The function updates the
    * second count, assigning the bound on the minutes.
    * Arguments: n is the timer number 0..79, 0..1..9
    * min is the number of minutes
    * sec is the number of seconds
    * frac is the number of 1/10 second
    * Returns:  none
    
    ***************************************************************
    */

    void TMRR_SETUP(int n, int min, int sec, int frac)
    {
      int pm;

      if (n < TMRR_MIN
        pm = STMP(n);
        TRM = TRM - pm^72(n);
        TMRR_COUNT(n);
        pm = pm^72(n);
        pm = pm^71(n);  /* record the current */
        return;
      }

      /*
      */
    /*STOP*/
Listing 7.1 (continued) TMR.C

/******************************************************************************
 * Description: Set the timer with the specified number of 1/10 seconds.
 * Parameters: "n" is the timer number 0.._TMR_MAX_TIMER
 * tcount is the number of 1/10 second to load into the timer
 * Source: None.
 ******************************************************************************/

void SetTimer(unsigned long n, unsigned int tcount)
{
    timer[n].tcount = tcount;
    timer[n].flags  = 0;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: SIGNAL TMR requires NOTICE THAT A "CLK TICK" HAS OCCURRED
 * Arguments: None.
 * Source: None.
 * Notes: _TMR_CLK_TICK must be set to provide 1/10 second delay.
 *        This can be set to _TMR_CLK_TICK/.10 if you use C/OS-II.
 ******************************************************************************/

void TmSignal(int n)
{
    _TMR_CLK_TICK = 0;
    _TMR_CLK_TICK = _TMR_CLK_TICK / 10;
}

/******************************************************************************
 * Description: Get the current value of the timer.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned long GetTimer(unsigned int n)
{
    return timer[n].tcount;
}

/******************************************************************************
 * Description: Get the number of timers.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned long GetNumberOfTimers()
{
    return _TMR_MAX_TIMER;
}

/******************************************************************************
 * Description: Get the flag values for the timer.
 * Parameters: "n" is the timer number 0.._TMR_MAX_TIMER
 * Source: None.
 ******************************************************************************/

unsigned int GetTimerFlags(unsigned int n)
{
    return timer[n].flags;
}

/******************************************************************************
 * Description: Set the timer flags.
 * Parameters: "n" is the timer number 0.._TMR_MAX_TIMER
 * flags is the timer flags to set.
 * Source: None.
 ******************************************************************************/

void SetTimerFlags(unsigned int n, unsigned int flags)
{
    timer[n].flags = flags;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: Start the timer.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void StartTimer(unsigned int n)
{
    timer[n].timer = 0;
    timer[n].flags = _TMR_CLOCKED;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: Stop the timer.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void StopTimer(unsigned int n)
{
    timer[n].flags = 0;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: Begin an interrupt handler.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void TmBeginInterrupt()
{
    _TMR_SET_INTERRUPT();
    m_interrupts = 1;
    _TMR_SET_INTERRUPT();
}

/******************************************************************************
 * Description: End an interrupt handler.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void TmEndInterrupt()
{
    _TMR_SET_INTERRUPT();
    m_interrupts = 0;
    _TMR_SET_INTERRUPT();
}

/******************************************************************************
 * Description: Enable interrupts for the timer.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void TmEnableInterrupts()
{
    _TMR_SET_INTERRUPT();
    _TMR_SET_INTERRUPT();
    m_interrupts = 1;
    _TMR_SET_INTERRUPT();
    _TMR_SET_INTERRUPT();
}

/******************************************************************************
 * Description: Disable interrupts for the timer.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void TmDisableInterrupts()
{
    _TMR_SET_INTERRUPT();
    _TMR_SET_INTERRUPT();
    m_interrupts = 0;
    _TMR_SET_INTERRUPT();
    _TMR_SET_INTERRUPT();
}

/******************************************************************************
 * Description: Get the timer number.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned int GetNumTimers()
{
    return _TMR_MAX_TIMER;
}

/******************************************************************************
 * Description: Get the timer status.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned int GetTimerStatus()
{
    return timer[n].status;
}

/******************************************************************************
 * Description: Set the timer status.
 * Parameters: "n" is the timer number 0.._TMR_MAX_TIMER
 * status is the timer status to set.
 * Source: None.
 ******************************************************************************/

void SetTimerStatus(unsigned int n, unsigned int status)
{
    timer[n].status = status;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: Get the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned int GetTicksPerSecond()
{
    return _TMR_CLK_TICK;
}

/******************************************************************************
 * Description: Set the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void SetTicksPerSecond(unsigned int n)
{
    _TMR_CLK_TICK = n;
}

/******************************************************************************
 * Description: Get the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned int GetTickCount(unsigned int n)
{
    return timer[n].tcount;
}

/******************************************************************************
 * Description: Set the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void SetTickCount(unsigned int n, unsigned int tcount)
{
    timer[n].tcount = tcount;
    _TMR_SET_TIMER(n);
}

/******************************************************************************
 * Description: Get the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

unsigned int GetTickCount()
{
    return _TMR_CLK_TICK;
}

/******************************************************************************
 * Description: Set the number of ticks per second.
 * Parameters: None.
 * Source: None.
 ******************************************************************************/

void SetTickCount(unsigned int n)
{
    _TMR_CLK_TICK = n;
}
Listing 7.1 (continued) 

/*
 *
 * DESCRIPTION: This function starts a timer.
 * ARGUMENTS:   Name is the timer to start.
 * RETURN:      None.
 */

void /*Start (THR2) */
{
   if (n == THR2) {
      OS_INTD(sleep);  //
      PROG(madel, 74.96)
      OS_INTD(sleep);
   }
}

/*STOP*/

/*
 * DESCRIPTION: This function stops a timer.
 * ARGUMENTS:   Name is the timer to stop.
 * RETURN:      None.
 */

void /*Stop (THR2) */
{
   if (n == THR2) {
      OS_INTD(sleep);  //
      PROG(madel, 74.96)
      OS_INTD(sleep);
   }
}

/*STOP*/
Listing 7.1 (continued)  TMR.C

static void TmrTask(void *data);

/*
  Description: This task is created by UserInit() and is responsible for updating the timers.
  The TmrTask() executes every 1/10 of a second.
  */
  Arguments: data (of type TMR微软雅黑_t);
  Return: None.
  */

static void TmrTask(void *data);

// Timer task

void (*pfunc)(void *);

/* Function to execute when timer times out */

void (*parg);

/* Arguments to pass to above function */

data = data;

/* Avoid compiler warning for unused arg. */

pfunc = NULL;

/* Start off with no function to execute */

parg = NULL;

for (i = 0; i < TMR_MAX_CNT; i++)

/* Decrement timer only if it is enabled */

if (pfunc = NULL;

/* No function to execute */

parg = NULL;

/* No argument to pass */

if (pfunc;

/* See if function expired */

parg;

/* See if expired */

parg;

/* Get pointer to function to execute */

/* ... and its argument */

if (pfunc;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

parg;

/* See if we need to execute function for */

parg;

/* ... timed out timer. */

pargs;
Listing 7.2  TMR.H

`#define TMRINIT_TIMEOUT 10000000 _HIC_ / 10`
`#define TMR_TMR1_TMR0 40`
`#define TMR_TMR1_TMR0 512`
`#define TMR_TMR0 10`
`#define TMR_TMR0 0`

`typedef TMR_TMR1_TMR0 TMR
#define TMR_DWT
#define TMR_DWT
#define TMR_DWT 0`

`/*
 DATA TYPES
 */`

`typedef struct { /* `}
 DECLARE TMR;
 DECLARE TMR_TMR1_TMR0;
 DECLARE TMR_DWT;
 void __Directive { /* Arguments supplied to user defined function */
 TMR_TMR1_TMR0;
 };`
Listing 7.2 (continued)  TMR.H

#include <stdlib.h>

/*
 * Table of timers managed by this module
 *
*/

extern void

/*
 * FUNCTION PROTOTYPES
 *
*/

void start(void);
void stop(void);

// Other function prototypes...

#define TMR görüyor(255, 255, 255)
Chapter 8

Discrete I/Os

Discrete inputs and outputs (I/Os) are found in most control and/or monitoring systems. The word discrete refers to the fact that the value taken by the input can take only one of two states. For example:

- 1 or 0
- TRUE or FALSE
- ON or OFF
- ENABLED or DISABLED
- PRESENT or ABSENT
- and so on.

**Figure 8.1 Discrete inputs.**

As shown in Figure 8.1, discrete inputs are generally used to monitor the state of manual switches, pressure switches (pressure exceeded or not), temperature switches (temperature exceeded or not), limit switches (device has reached its limit or not), relay contact closures (open or closed), proximity detectors
Discrete inputs are generally used to determine the state of an input. In some applications, however, you need to know whether a discrete input has changed state or not and, possibly, how many times it did so.

Discrete outputs are used to control lamps, relays, fans, alarms, heaters, valves, etc. (See Figure 8.2.) A discrete output is generally either in one state or the other. A blinking light versus a light that is always ON, however, does a better job of attracting the attention of a user to an error condition.

![Discrete Outputs Diagram](image)

In this chapter, I will provide you with a module that monitors discrete inputs and controls discrete outputs. The module allows you to have as many discrete inputs and outputs as you need (up to 20 each). For each discrete input, you will be able to:

- Determine whether the input is 1 or 0.
- Determine whether a transition from 1 to 0 or from 0 to 1 occurred on the input.
- Determine how many transitions from 1 to 0 or from 0 to 1 occurred on the input.
- Simulate a toggle switch with a momentary closure switch.
- Bypass the hardware for debug purposes.

For each discrete output, you will be able to:

- Turn the output ON or OFF.
- Blink the output at a user-definable rate (one for each output).
- Bypass your application code to control the output during debugging.

8.00 Discrete Inputs

Reading discrete inputs is a fairly trivial task. You need only provide your microprocessor with as many parallel input lines as you have discrete inputs to read. The microprocessor simply needs to read the input ports, mask off unwanted inputs, and make a decision based on the state of the input.

I generally prefer to put a layer of software between my application code and the hardware so I can change the hardware without affecting the application software. Putting a layer of software also allows you to test your application before you get your hands on the hardware. I like to give a logical address
to each discrete input, typically from 0 to n. You can thus write a simple function that returns the state of any logical discrete input to your application as shown in the following pseudocode:

```c

BOO JEN DIGES (DDUPU n)
{
    Read port where discrete input to is located;
    Mask off unwanted bits;
    return the state of the discrete input (either TRUE or FALSE);
}
```

The mask is an 8-bit value that selects the desired bit to read. For example, to read the state of bit 4 (bits are numbered 0 to 7 from right to left), the mask would be 00100000. With such a function, your code will be a little bit slower and your code size will increase but the benefits are enormous. Now you can change the hardware as many times as you need and your application code will never know the difference. By encapsulating access to the hardware we can also handle cases where some of the inputs are inverted by the hardware and still return the proper state to the application code. In other words, if an input is considered a logical 0 when it is HIGH, then DIGES() can invert the value of the input read and report a 0 to the application code.

If you have spare address space and a "say" about hardware design, you should consider using one of my favorite chips for discrete inputs: the 74251 8-input data selector/multiplexer, shown in Figure 8.3. Note that you can have as many discrete inputs as needed by simply adding 74251s.

![Figure 8.3 Discrete inputs using 74251.](image)

Basically, each discrete input has its own address in the microprocessor address space. Reading a discrete input becomes trivial:

```c

BOO JEN DIGES (DDUPU n)
{
    return (Read value from address of port 'n' and mask with 0x0f);
}
```
Even with `DIGet()`, it is still up to your application to determine whether a discrete input has changed state. To determine if an input has changed state, you will need to repeatedly call `DIGet()` (i.e., poll the input) and compare the previous value with the current one. The input has changed state when both values are different. If you need to know whether the input changed from 0 to 1, you will further need to add code to ensure that the previous state was 0.

What if you had a momentary closure switch connected to a discrete input and needed to simulate a toggle switch? (That is, you press the switch once to turn a device ON and you press the switch again to turn the device OFF.) To accomplish this, you need to change the state of a variable whenever a transition from 0 to 1 is detected.

The discrete I/O module presented in this chapter allows you to configure any discrete input to handle all of the situations described earlier. Each discrete input is considered a *logical channel*. The discrete I/O module allows you to have as many logical channels as you need (up to 256). Figure 8.4 shows a flow diagram of a *discrete input channel*. Note that I used electrical symbols to represent the functions performed by each discrete input channel. Of course, all functions are handled in software.

**Figure 8.4 Discrete input channel.**
Chapter 8: Discrete I/Os — 259

As Figure 8.4 shows, each discrete input channel has the capability to be configured (at run-time) to any of nine modes through the Mode Select Switch:

1. Always return a 0.
2. Always return a 1.
3. Return the state of the hardware input.
4. Return the complement of the hardware input.
5. Detect negative-going transitions and return the number of transitions detected.
6. Detect positive-going transitions and return the number of transitions detected.
7. Detect both positive- and negative-going transitions and return the number of transitions detected.
8. Toggle between 0 and 1 when a negative transition is detected.
9. Toggle between 0 and 1 when a positive transition is detected.

To reduce the code size of your application, the edge detection features can be disabled at compile time, as shown in Figure 8.4.

To provide the functionality described earlier, all discrete inputs are read and processed on a continuous basis. In other words, all inputs are polled. Because of this, the maximum rate at which discrete inputs can change state is based on how often inputs are polled. Polling is handled by a task (described later) which executes at a regular interval (you decide at compile time how often the task will execute).

Discrete inputs must not change state any faster than half the task execution rate of the discrete I/O module. That is, the task must execute twice as fast as the expected rate of change of discrete inputs.

Your application knows about discrete input channels through interface functions. The interface functions allow you to set the configuration mode of each channel through the Mode Select Switch, set the state of the Bypass Switch and, if the bypass switch is open, bypass the hardware. Bypassing of the hardware is accomplished by having an interface function deposit a value into the discrete channel.

Where your application is concerned, it doesn’t know that the value received didn’t come from the actual hardware.

8.01 Discrete Outputs

Updating discrete outputs is a straightforward operation but a little trickier than updating discrete inputs. All you need is to provide your microprocessor with enough latched parallel output lines so you have discrete outputs to control. As with discrete inputs, I generally prefer to put a layer of software between my application code and the hardware. This prevents the application code from knowing what kind of hardware is involved and how it is accessed. I can thus port my application code to other environments by simply changing the hardware interface functions. I give a logical address to each discrete output, typically from 0 to n. For discrete outputs connected to an 8-bit latched parallel output port, you have two scenarios: either you can read back the contents of the output port (intel 8255A or Motorola 6821) or else the port is write-only (74273, 74373, etc.). This pseudocode for a port that can be read back would look like this:
void D0Get(INPORT n, BOOLEAN val)
{
    Disable interrupts;
    Read the output port;
    if (val == FALSE) {
        AND the port data with complement of 'mask';
    } else {
        OR the port data with mask;
    }
    Write new data to port;
    Enable interrupts;
}

The mask is an 8-bit value that selects the desired bit to set or clear. For example, to set or clear bit 6 (bits are numbered 0 to 7 from right to left), the mask would be $0x40$. Note that you also need to disable interrupts because updating the discrete output is considered a critical section. Forgetting to disable interrupts is a common mistake. The pseudo-code for a port that cannot be read back follows this paragraph. In this case, an image of the output port's content is maintained in memory (i.e., RAM).

void D0Set(INPORT n, BOOLEAN val)
{
    Disable interrupts;
    if (val == FALSE) {
        AND the memory image with the complement of the 'mask';
    } else {
        OR the memory image with the mask;
    }
    Write memory image to port;
    Enable interrupts;
}

If you have spare address space and a "say" about hardware design, you should consider using one of my favorite chips for discrete outputs: the 74259 8-bit addressable latch, as shown in Figure 8.5. Note that you can have as many discrete outputs as needed by simply adding 74259s.
Figure 8.5  Discrete outputs using 74259.

<table>
<thead>
<tr>
<th>74259</th>
<th>Discrete Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td></td>
</tr>
<tr>
<td>WR</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>C</td>
</tr>
<tr>
<td>A1</td>
<td>B</td>
</tr>
<tr>
<td>A0</td>
<td>A</td>
</tr>
</tbody>
</table>

Basically, each discrete output has its own address in the microprocessor address space. Updating a discrete output becomes trivial:

```c
void DOSet(n, BOOLEAN val)
{
    Output value to address of port 'n';
}
```

What if you needed to blink one or more discrete outputs? Blinking outputs are quite useful when connected to lights because they can be used to signal alarm conditions to users. To blink an output, you could call `DOSet()` to change the state of an output at a regular interval from your application code. This obviously complicates your application.

The discrete I/O module presented in this chapter allows you to control discrete outputs and also blink any (or all) of the discrete outputs.

Each discrete output is considered a logical channel. The discrete I/O module allows you to have as many logical channels as you need (up to 256). Figure 8.6 shows a flow diagram of a discrete output channel. Note that I used electrical symbols to represent the functions performed by each discrete output channel. Of course, all functions are handled in software.
Figure 8.6  Discrete output channel.

As shown in Figure 8.6, each discrete output channel has the capability to be configured (at runtime) to any of five modes (through the Mode Select Switch):

1. Always output a 0.
2. Always output a 1.
3. Directly output what your application desires to put out.
4. Blink the output asynchronously (described below).
5. Blink the output synchronously (described below).

Your application software can also complement (or invert) the output through the Invert Select Switch.

If either of the two blinking modes is selected, your application can determine whether blinking will be enabled through the Blink Enable Select Switch. To reduce the code size of your application, the blinking feature can be disabled at compile time, as shown in Figure 8.6.

Your application knows about discrete output channels through interface functions. The interface functions allow you to:

- Set the configuration mode of each channel through the Mode Select Switch.
- Set the Blink Enable Select Switch, which determines how to enable blinking.
- Determine whether the output will be inverted by setting the Invert Select Switch.
- Set the blinking rate by specifying the values for A, B, and C (see Figure 8.6).
- Set the state of the Bypass Switch and, if the bypass switch is open, bypass your application code.

Bypassing of your application is accomplished by having an interface function deposit a value into the discrete channel. Where your application is concerned, it still thinks it is controlling the discrete output channel.

When you choose to blink a discrete output, you need to specify the type of blinking: either asynchronous or synchronous. In asynchronous mode, you need to specify the duty cycle through two
variables: A (the ON time) and B (the total time). Because each discrete output can have different A and B values, blanks occur asynchronously. In asynchronous mode, you specify the ON time (variable A) with respect to a common (to all asynchronous discrete outputs) total time (variable C). The ON time and total time are based on how often the discrete I/O module executes. If the discrete I/O module executes 10 times per second then, an ON time of one second requires A to be set to 10.

8.02 Discrete I/O Module

The source code for the discrete I/O module is found in the `\SOFTWARE\BLOCKS\DIO\SOURCE` directory. The source code is found in the files DIO.c (Listing 8.1) and DIO.h (Listing 8.2). As a convention, all functions and variables related to the discrete I/O module start with either DIO (functions or variables common to both discrete input and outputs), DI (discrete input functions or variables), or DO (discrete output function or variables). Similarly, #define constants will either start with DIO_, DI_, or DO_.

8.03 Discrete I/O Module, Internals

Figure 8.7 shows a flow diagram of the discrete I/O module. (You can also refer to Listings 8.1 and 8.2 for the following description.) The discrete I/O module consists of a single task (DIOTask()) that executes at a regular interval (DIO_TASK_MONOS). DIOTask() can manage as many discrete inputs and outputs as your application requires (up to 250 each). The discrete I/O manager is initialized by calling DIOInit(). Every DIO_TASK_MONOS, DIOTask() calls DIOInit(), DIOUpdate(), and DOWR().
Figure 8.7 DIO module flow diagram.

DITbl[] is a table that contains configuration and run-time information for each discrete input channel. An entry in DITbl[] is a structure defined in DIO.h and is called DIO_DI. Discrete inputs are read and mapped to DITbl[] by the hardware interface function DIRD(). DIRD() knows about your hardware and thus can be easily changed to adapt to your environment.

Figure 8.8 shows a flow diagram of a discrete input channel. Note that I used electrical symbols to represent functions performed in software for each discrete input channel. DITIn, DIModeCtrl, DIBypassEn, and DIOVal are structure members of DIO_DI (see DIO.h). DIOUpdate() is responsible for updating all the discrete input channels. Discrete input channels that are configured for edge detection are processed by DITrigger(). DITrigger() keeps track of the previous state (DIFree) of the discrete input and is used to determine if an input has changed state.
Figure 8.8  Discrete input channel.

DOUT(): is a table that contains configuration and run-time information for each discrete output channel. An entry in DOUT[] is a structure defined in IIO.h and is called DIO[0]. Discrete outputs are mapped from DOUT[] to your hardware through the interface function DOUT().

DIN[] knows about your hardware and this can be easily changed to adapt to your environment.

Figure 8.9 shows a flow diagram of a discrete output channel. Note that I used electrical symbols to represent functions performed in software for each discrete output channel. .DOUT[], .DOByPass, .DOByPassEnd, .DOByPassDelay, .DOByPassDelayEnd, and .DOByPass are structure members of DIO[0] (see DOUT). .DOByPass() is responsible for updating all the discrete output channels.
Figure 8.9 Discrete output channel.

As previously mentioned, there are two blinking modes: synchronous and asynchronous. Synchronous blinking mode is shown in figure 8.10. When a discrete output channel is in this mode, its output is HIGH (or LOW depending on the state of .DOInv) when .DOA is less than .DOsynCtr. .DOsynCtr counts from 0 to .DOsynCtrMax (set by .DOSetSynCtrMax()). .DOsynCtr is cleared when it reaches .DOsynCtrMax. This mode is synchronous because all discrete output channels in this mode are referenced to .DOsynCtr.

Figure 8.10 Synchronous blinking mode.

Asynchronous blinking mode is shown in Figure 8.11. When a discrete output channel is in this mode its output is HIGH (or LOW depending on the state of .DOInv) when .DOA is less than .DOOff.
8.04 Discrete I/O Module, Interface Functions

Your application software knows about the discrete I/O module through the interface functions shown in Figure 8.12.

Figure 8.12 Discrete I/O module interface functions.
To allow the code size in your application to be reduced, I have added two defines, which are used to enable/disable code generation for edge detection for discrete inputs (DI_EDGE_INPUT) and enable/disable code generation for blinking of discrete outputs (DI_BLINK_OUTPUT). Setting these #defines to 1 will enable code generation for the respective code.
DICfg2EdgeDetectFunct()
    void DICfg2EdgeDetectFunct(uint8_t, void (*)(void), void *arg);

When a discrete input channel is configured for edge detection and a transition is detected, a user-definable function can be executed. The function to execute is specified to the discrete input channel by calling DICfg2EdgeDetectFunct().

Arguments
- n is the discrete input channel you wish to configure. Discrete input channels are numbered from 0 to DIS_MAX_DT - 1.
- fct is a pointer to the function that will be executed whenever a transition is detected. Note that passing a NULL pointer indicates that no function is to be executed when a transition is detected. All discrete input channels have NULL pointers by default. When the function is called, it is passed a pointer to void (i.e., the arg). This allows for different arguments to be passed to a reentrant function. You must declare the function that will be called as follows:

  void UserFunct (void *arg);

Note that UserFunct() is called with the argument that you specify in DICfg2EdgeDetectFunct(), that is, arg. This allows you to design a single function that can be used by more than one discrete input channel. The user-defined function will be called by the discrete I/O manager task DISEvent() when a transition is detected on the input. The execution time of the discrete I/O task is augmented by the execution time of all the functions that will execute when a transition is detected in their respective inputs.

Return Value
None

Notes/Warnings
Some applications do not require the execution of a function upon detection of a transition. In these situations, there is no need to call DICfg2EdgeDetectFunct() because the initial value of the pointer to a function for each discrete input channel is NULL. In other words, the discrete I/O task will not execute any function when pointing to NULL.
Example

The function that executes when a transition is detected can signal another task through a semaphore, a mailbox, or even a message queue. This would allow you to defer processing of input transition detection to either a lower- or higher-priority task.

```c
void Task (void *pdata)
{

    DIEvent *DIEvent;

    DIEvent = DIEventCreate(0);  
    DITypedSeq(0, DL_WORKER_FACE_HIGH_QOIND));  
    DIFaceInitProc(0, DIEventProc, (void *)DIEvent);
    for (;;) {
        DIEventWait(DIEvent, 0, &err); /* Wait for DI to transition */
    }

    DIEventRelease(DIEvent, 0, &err); /* DI transitioned */
}
```

```c
void DIEventProc (void *arg)
{
    DIEventProc(DI_EVENT *arg); /* DI transitioned */
}
```
DICfgMode()  

void DICfgMode(int mode);  

DICfgMode() is used to set the operating mode of a discrete input channel.

Arguments

mode determines the operating mode of the discrete input channel. The discrete I/O module currently supports nine modes:

1. DI_MODE_LOW allows DIGet() to always return 0. This function basically simulates grounding an input.
2. DI_MODE_HIGH is similar to DI_MODE_LOW in that it allows DIGet() to always return 1. This function basically simulates tying an input high.
3. DI_MODE_INVERT allows the discrete input channel to read whatever is present on the hardware input. This is the default mode for a discrete input channel.
4. DI_MODE_INN allows the discrete input channel to read the complements of whatever is present on the hardware input.
5. DI_MODE_EDGE_LOW_OHNG allows the discrete input channel to detect and count transitions from 0 to 0 on the hardware input. The frequency of the input signal must be less than the scan rate of the discrete I/O module (determined by DIG_TASK биз task). DIGet() will return the number of 0 to 0 transitions detected. Note that the number of transitions can be cleared by calling DIClr() (described later).
6. DI_MODE_EDGE_HIGH_OHNG allows the discrete input channel to detect and count transitions from 0 to 1 on the hardware input. The frequency of the input signal must be less than the scan rate of the discrete I/O module. DIGet() will return the number of 0 to 1 transitions detected. Note that the number of transitions can be cleared by calling DIClr() (described later).
7. DI_MODE_EDGE_BOTH allows the discrete input channel to detect and count either transitions from 0 to 1 or from 1 to 0 on the hardware input. The transition rate of the input signal must be less than the scan rate of the discrete I/O module. DIGet() will return the number of transitions detected. Note that the number of transitions can be cleared by calling DIClr() (described later).
8. DI_MODE_TOGGLE_LOW_OHNG allows the state of the discrete input channel to change whenever a transition from 1 to 0 is detected. Again, the transition rate of the input signal must be less than the scan rate of the discrete I/O module.
9. DI_MODE_TOGGLE_HIGH_OHNG allows the state of the discrete input channel to change whenever a transition from 0 to 1 is detected. Again, the transition rate of the input signal must be less than the scan rate of the discrete I/O module.
Return Value
None

Notes/Warnings
None

Example

void main (void)
{

  D1CfgMode(0, D1_MODE_DIRECT);

}

Chapter 6: Discrete I/Os — 273

**DIClir**

```c
void DIClir(u8 n);
```

The only way to clear the number of transitions detected when the discrete input channel is configured for edge detection is to call `DIClir()`. The function has no effect if the channel is not configured for edge detection.

**Arguments**

- `n` is the discrete input channel you wish to clear. Discrete input channels are numbered from 0 to `DI0_MAX_DI - 1`.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task(void *pData)
{
    DICfgmode(DI, DI_MODE edge, RISE, GND);
    for (;;) {
        ...
        DIClir(0); /* Clear the number of transitions of channel 0 */
        ...
    }
}
```
DIGet ( )

The current value of the discrete input channel can be obtained by calling DIGet ( ). If the discrete input channel is configured for edge detection, the returned value will correspond to the number of transitions detected by the channel. If the discrete input channel is not configured for edge detection, the returned value will either be 0 or 1.

Arguments

n is the discrete input channel you wish to read. Discrete input channels are numbered from 0 to DIO_MAX_DI - 1.

Return Value

The current value of the discrete input channel or the number of transitions.

Notes/Warnings

None

Example

```c
void task (void *data)
{
    INITDI transitions;

    MCIgMode(0, D1 Mode_DIR_EDS_EDGE_EN)
    for (i=0; i < 10; i++)
    {
        transitions = DIGet(0); /* Get number of transitions on DI #1 */
    }
}
```
**DIOInit()**

```c
void DIOInit(void);
```

DIOInit() is the initialization code for the discrete I/O module. DIOInit() must be called before you use any of the other discrete I/O module functions. DIOInit() is responsible for initializing the internal variables used by the module and for the creation of the task that will update the discrete inputs and outputs.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void main (void)
{
    ...
    DIOInit();
    ...
}
```
DISetBypass()  

void DISetBypass(DINTU n, DINT16U val);  

Your application software can bypass or override the discrete input channel value by using this function. 
DISetBypass() doesn’t do anything unless you have opened the bypass switch by calling DISetByp-
assEn() as described earlier.

Arguments  

n is the discrete input channel you wish to bypass. Discrete input channels are numbered from 0 to
DIN_MAX_DI – 1.

val is the value you want DIGet() to return to your application. Because val is a DINT16U, you can
set the number of transitions detected when the discrete input channel is configured for edge detection.

Return Value  

None

Notes/Warnings  

None

Example

void Task (void *pdata)  
{
    for ( ; ; ) {
        DISetBypassEn(0, TRUE); /* Bypass channel #0 */
        DISetBypass(0, 1);       /* Set value of channel #0 */
    }
}
**DISetBypassEn()**

void DISetBypassEn(DIO n, BOOLEAN state);

DISetBypassEn() allows your application code to prevent the 'physical' discrete input channel from being updated. This permits your application to set the value returned by DISet(). The value of the discrete input channel's set by DISetBypass(). DISetBypassEn() and DISetBypass() are very useful for debugging.

**Arguments**

- **n** is the discrete input channel you wish to bypass. Discrete input channels are numbered from 0 to DIS_MAX_DI - 1.
- **state** is the state of the bypass switch. When TRUE, the bypass switch is open (i.e., the discrete input channel is bypassed). When FALSE, the bypass switch is closed (i.e., the discrete input channel is not bypassed).

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *pdata)
{
   for (;;) {
      .
      .
      DISetBypassEn(0, TRUE); /* Bypass channel */
      .
   }
}
```
**DOCfgBlink()**

```c
void DOCfgBlink(int16_t a, int16_t mode, int16_t a, int16_t b);
```

**DOCfgBlink()** allows you to configure the discrete output blinking mode.

**Arguments**

- a is the discrete output channel you wish to configure for blink mode. Discrete output channels are numbered from 0 to DIGI_MAX.DO – 1.
- mode sets the state of the Blink Enable Select Switch to one of three values:
  1. DO_ENABLE_0N allows the discrete output to blink continuously.
  2. DO_ENABLE_0FF allows the discrete output to blink only if the input to the discrete output channel is set to 1. Blinking stops when the input to the discrete output channel is set to 0. In this case, the output is forced LOW unless it’s inverted.
  3. DO_ENABLE_0F_F allows the discrete output to blink only if the input to the discrete output channel is set to 0. Blinking stops when the input to the discrete output channel is set to 1. In this case, the output is forced LOW unless it’s inverted.
- a specifies the ON time for either synchronous or asynchronous mode (the A value in Figures 8.9, 8.10, and 8.11). The actual ON time is determined by the execution rate of the discrete I/O module. a is given by:

  \[ a = \text{ON time (sec.)} \times \text{Task execution rate (Hz)} \]

- b specifies the total period when the discrete output is configured for asynchronous mode (the B value of Figures 8.9 and 8.11). The period is determined by the execution rate of the discrete I/O module. b is given by:

  \[ b = \text{Period (sec.)} \times \text{Task execution rate (Hz)} \]

**Return Value**

None

**Notes/Warnings**

None
void Task (void *pdata)
{
    DOCfgShLink(0, DO_CFG_LNK_BNK, 10, 20);
    for (;;) {
    ...
    ...
    }
}
void DOCfgMode(INTRU n, int8u mode, BOOLAM inv);

DOCfgMode() is used to set the operating mode of a discrete output channel. Each channel must be individually configured.

**Arguments**

- \( n \) is the desired discrete output channel to configure. Discrete output channels are numbered from 0 to \( DTO\_MAX\_DO = 1 \).

- \( \text{mode} \) determines the operating mode of the discrete output channel. The discrete I/O module currently supports five modes:
  1. DO_MODE_LO = the default mode and forces the discrete output LOW.
  2. DO_MODE_HI = is similar to DO_MODE_LO, except that it forces the discrete output HIGH.
  3. DO_MODE_DIRECT allows the discrete output channel to output whatever state you set through DOSet() or DOSetByPass().
  4. DO_MODE_BLINK_SYNC allows the discrete output to continuously change from LOW to HIGH and from HIGH to LOW. In this mode, you also need to specify how long the output will be HIGH with respect to a continuously running counter, DOSyncCounter, which is specified through DOSyncCounterMax(). If DOSyncCounter is allowed to count from 0 to 100 then, to get a 25 percent duty-cycle, you need to set the HIGH time to 25. This is done by calling DOCfgBlkLink().
  5. DO_MODE_BLINK_ASYNC allows the discrete output to continuously change from LOW to HIGH and from HIGH to LOW. In this mode, you also need to specify how long the output will be HIGH and the total period of the signal. This is done through DOCfgBlkLink().

- \( \text{inv} \) is used to complement the output. When \( \text{inv} \) is set to \text{TRUE}, the output is complemented as shown in Figure 8.9.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void main(void)
{

    DOCfgMode(0, DO_MODE_BLINK_SYNC, FALSE);
}
```
**DOGet()**

**BOOLEAN DOGet(DRWUS n);**

`DOGet()` allows your application to get the state of the output that actually goes to the hardware. `DOGet()` returns either TRUE (the output is set to 1) or FALSE (the output is set to 0).

**Arguments**

`n` is the discrete output channel you wish to monitor. Discrete output channels are numbered from 0 to `DO_MAX_DO - 1`.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *pdata) 
{
    BOOLEAN state;

    for (;;) 
    {
        state = DOGet(0); /* Get value of channel #0 */
    }
}
```
DOSet()

void DOSet(DIOU u, BOOLEAN state);

DOSet() allows your application to set the state of the discrete output channel. If the discrete output channel is configured for blink mode, the state passed to DOSet() is used to enable or disable blinking, as shown in Figure 8.9.

Arguments

u is the discrete output channel you wish to set. Discrete output channels are numbered from 0 to DIO_MAX_DIO - 1.

state is the desired state of the discrete output, and can be either TRUE or FALSE. Note that the state of the discrete output occurs before any processing is performed on the discrete output channel, as shown in Figure 8.3.

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    for (;;) {
        
        DISetBypass(0, 1); /* Set value of channel 0's .DIVal */
        
    }
}
Chapter 8: Discrete I/Os — 283

**DOSetBypass()**

```
void DOSetBypass(DWORD n, BOOLEAN state);
```

You can bypass what your application code is sending to the discrete output channel by using this function. `DOSetBypass()` doesn't do anything unless you have opened the bypass switch by calling `DOGetBypassEnable()`, as described earlier.

**Arguments**

- `n` is the desired discrete output channel to override. Discrete output channels are numbered from 0 to `DIO_MAX_DO - 1`.
- `state` is the desired state of the discrete output and can be either `TRUE` or `FALSE`. Note that the bypass occurs before any processing is performed on the discrete output channel, as shown in Figure 8.9.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *Update)
{
    for (i = 0; i < ...
        DOSetBypass(0, 1);  /* Set value of channel #0's .DIOval */
    ...
}
```
DOSetBypassEn()

void DOSetBypassEn(DIO n, BOOLEAN state);

DOSetBypassEn() allows your application code to bypass your application and set the state of the discrete output by calling DOSetBypass(). DOSetBypassEn() and DOSetBypass() are very useful for debugging.

Arguments

n is the desired discrete output channel to bypass. Discrete output channels are numbered from 0 to DIO_MAX_DO - 1.

state is the state of the bypass switch. When TRUE, the bypass switch is open (i.e., the discrete output channel is bypassed). When FALSE, the bypass switch is closed (i.e., the discrete output channel is not bypassed).

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    for (i) {
        ...
        DOSetBypassEn(i, TRUE); /* Bypass channel */
    }
}
DOSetSyncCtrMax() is used to set the period for the synchronous blinking mode. The synchronous blinking mode is useful when you need to have lights blink at the same rate.

**Arguments**

`val` specifies the total period when the discrete output is configured for synchronous mode (the C value of Figures 8.9 and 8.10). The period is determined by the execution rate of the discrete I/O module. `val` is given by:

\[
val = \text{Period (sec.)} \times \text{Task execution rate (Hz)}
\]

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void Task (void *pdata)
{
  DOSetSyncCtrMax(100);
  ...;
}
```
8.05 Configuration

I added two #defines, DIO_EDGE_EN and DIO_BLINK_MOSI_EN, which are used to enable/disable some of the functionality of the discrete I/O module in order to reduce the amount of ROM and RAM. Specifically, DIO_EDGE_EN allows you to remove edge detection for all discrete input channels, and DIO_BLINK_MOSI_EN allows you to remove the blinking capability of discrete output channels.

You could reduce the amount of RAM for each discrete input or output by using bit fields in the DIO_DI and DIO_DO structures. In this case, you would reduce the amount of RAM required at the expense of more code space (manipulation of bit fields requires more code and is slower).

Configuring the discrete I/O module is fairly simple.

1. You need to define the value of some #defines. The #defines are found in DIO.h and CFG.h.

   WARNING
   In the previous edition of this book, DIO_TASK_STACK_SIZE specified the size of the stack for DIO_TASK() in bytes. μC/OS-II assumes the stack is specified in stack width elements.

2. You will need to adapt DIOInit(), DIOWr(), and DIOInitIO() to your specific environment.

   All physical discrete inputs are read by DIORead() and are mapped to their corresponding DIO_DI structures, as shown in Figure 8.13. In the code I provided in Listing 8.1, DIOInit() obtains its discrete inputs from an 8-bit parallel port. The least significant bit of the input port corresponds to discrete input channel #0, the next-to-the-least significant bit is channel #1, and so on. Adding more discrete inputs should be a trivial task.

**Figure 8.13 Mapping of physical inputs to discrete input channels.**

8-Bit Parallel Input Port

<table>
<thead>
<tr>
<th>B7</th>
<th>B6</th>
<th>B5</th>
<th>B4</th>
<th>B3</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
</table>
| DI1b[11]| DI1n
| DI1b[10]| DI1n
| DI1b[9] | DI1n
| DI1b[8] | DI1n
| DI1b[7] | DI1n
| DI1b[6] | DI1n
| DI1b[5] | DI1n
| DI1b[4] | DI1n

Figure 8.14 shows how discrete output channels are mapped to physical outputs using DOOut(). In the code provided in Listing 8.1, discrete output channels are mapped to an 8-bit parallel port. Discrete output channel #0 is mapped to the least significant bit of the output port (i.e., bit 0). Channel #1 is mapped to bit 1, and so on. Adding more discrete outputs should be fairly simple.
**Figure 8.14** Mapping of discrete output channels to physical outputs.

DIOInitIO() is the initialization code which is called by DIOInit() and is used to initialize your physical hardware port. For example, if you are using Intel's 82C55A Programmable Peripheral Interface (PPI), you would initialize the 82C55A to the desired mode in DIOInitIO().

### 8.06 How to Use the Discrete I/O Module

To use the discrete I/O module, you will need to call DIOInit() prior to using any of the other functions. You would typically do this in main() as follows:

```c
void main(void) {
    OSInit();        /* Initialize the O.S. (uC/OS-II) */
    .
    DIOInit();       /* Initialize the discrete I/O module */
    .
    OSStart();       /* start multitasking (uC/OS-II) */
}
```

Once you have initialized the discrete I/O module, you can configure each one of the discrete inputs and outputs by calling DIOCfgNode(), DIOCfgEdgeDetect(), DIOCfgdecode(), and DIOCfgBlind(). You will also need to call DIOSetSyncCtlMax() if you are using any of the discrete outputs.
in synchronous blink mode. You can choose to configure discrete I/O channels immediately after the call to DIOInit() or in your application task, as shown:

```c
void AppTask (void *data)
{
    data = data;
    /* Initialize discrete I/O channels here ...*/

    for (i = 0; i < 8; i++)
        /* Application task code ... */
}
```

A traffic light controller would be an ideal application for the discrete I/O module. For the intersection shown in Figure 8.15, you would need eight discrete outputs to control the state of each traffic light (four for North <-> South, four for East <-> West). Each set of four outputs would control:

- 1 green light
- 1 yellow light
- 1 red light
- 1 green light (for left turn arrow)

This traffic light controller caters to pedestrians. Two buttons are needed at each corner so pedestrians can request to cross the intersection. The controller, however, only needs to see two discrete inputs: one to request an East/West crossing and another to request a North/South crossing. Additional lights are required to inform the pedestrian when it is safe to cross the intersection: a walk light and a don't walk light. The don't walk typically blinks when it is no longer safe to cross the intersection. You will need four discrete outputs for pedestrian crossing lights.

Figure 8.16 shows a block diagram of the traffic light controller and the necessary discrete I/Os. The code required to configure the discrete I/Os for the traffic controller follows this paragraph. All discrete outputs are initially configured for direct mode. The mode of the discrete output controlling the don't walk light can be changed to blinking mode when it is unsafe to cross the street.
Figure 8.15 Traffic light control using the discrete I/O module.

Figure 8.16 Traffic light control block diagram.
void TrafficCtrlInit0(void)
{
    DCfgMode( 0, DO_MODE_EDGE_LOW_00160);  /* Pedestrian buttons */
    DCfgMode( 1, DO_MODE_EDGE_LOW_00230);

    DCfgMode( 0, DO_MODE_DIRECT);  /* Traffic lights */
    DCfgMode( 1, DO_MODE_DIRECT);
    DCfgMode( 2, DO_MODE_DIRECT);
    DCfgMode( 3, DO_MODE_DIRECT);
    DCfgMode( 4, DO_MODE_DIRECT);
    DCfgMode( 5, DO_MODE_DIRECT);
    DCfgMode( 6, DO_MODE_DIRECT);
    DCfgMode( 7, DO_MODE_DIRECT);

    DSet(1, ON);  /* Turn ON N/S Green light */
    DSet(7, ON);  /* Turn ON E/W Red light */

    DCfgMode( 8, DO_MODE_DIRECT);  /* Pedestrian lights */
    DCfgMode( 9, DO_MODE_DIRECT);
    DCfgMode(10, DO_MODE_DIRECT);
    DCfgMode(11, DO_MODE_DIRECT);

    DSet( 9, ON);  /* Turn ON "DON'T WALK" */
    DSet(11, ON);
}
Listing 8.1  DIO.C

/*
 * Embedded Systems Building Blocks
 * Complete and Ready-to-Use Modules in C
 * Discrete I/O Module
 * (C) Copyright 1999, Jane J. Lehtinen, Newton, MA
 * All Rights reserved
 *
 * Filename : DIO.C
 * Programmer : Jane J. Lehtinen
 */

/*
 * INCLUDE FILES
 */

#define DIOGLOBALS

#include "includes.h"

/*
 * LOCAL VARIABLES
 */

#define DIO_MAIN(argc, argv, envp)

main(argc, argv, envp) int argc; char *argv[]; char *envp[];

{ /*
 */

static char *DIO看見 = NULL;

static int DIO_main = 0;

DIO_main();

static void DIO_update(void);

static void DIO_read(void *DIO);

static void DIO_write(void *DIO);

"DIO.C"/
Listing 8.1 (continued)  DIO.C

/**
 * 
 * Description: This function is used to configure the edge detection capability of the discrete input channel.
 * 
 * Arguments: n is the discrete input channel to configure.  DIO_MAX (1).
 * 
 * pcf is a pointer to a function that will be executed if the desired edge has been detected.
 * 
 * arg is a pointer to arguments that are passed to the function called.
 * 
 * Returns: None.
 */

 const unsigned int DIO_MAX = 1;

 void DIO_edge(uint32_t n, void (*pcf)(void *), void *arg)
 {
     char DIO_edge_caller[] = __func__;
     DIO_edge_model = pcf;
     DIO_edge_arg = arg;
 }

 #endif

/**
 * 
 * Description: This function is used to configure the edge detection capability of the discrete input channel.
 * 
 * Arguments: n is the discrete input channel to configure.  DIO_MAX (1).
 * 
 * mode is the enabled mode and can be:
 * 
 * DIO_MODE_EDGE on input is forced low
 * DIO_MODE_EDGE on input is forced high
 * DIO_MODE_EDGE is based on state of physical sensor (default)
 * DIO_MODE_EDGE is based on complement of physical sensor
 * DIO_MODE_EDGE is both a low-going and a high-going transition are detected
 * DIO_MODE_EDGE is both a low-going and a high-going transition are detected in toggle mode
 * 
 * Returns: None.
 * 
 * Notes: Edge detection is only available if the configuration constant DIO_MODE is set to 1.
 */

 void DIO_edge(uint32_t n, void (*pcf)(void *), void *arg)
 {
     if (n < DIO_MAX)
     {
         DIO_edge_caller[] = __func__;
         DIO_edge_model = pcf;
         DIO_edge_arg = arg;
     }
 }

 #endif
Listing 8.1 (continued) DIO.C

/*
 * CLASSES & DEVICE PROXY
 *
 * Description: This function clears the number of edges detected if the discrete input channel is
 * configured to count edges.
 * Arguments: n is the discrete input channel (0..NLIN, NLIN+1) to clear.
 * Returns: none
 */

#define NLIN 8

void DIOClr (DIO_N VI n) {

    if (n < NLIN+2) { /* 0..NLIN, NLIN+1 */
        "DIR = 0x0000;"
        if (n == 0) { /* only clear EDGES */
            DIO_N I; /* See if edge detection mode selected */
            if (n == 0) { /* only clear EDGES */
                DIO_N I; /* See if edge detection mode selected */
            } /* Clear the number of edges detected */
        } /* Clear the number of edges detected */
    } /* Clear the number of edges detected */
}

} Result
"MSRSET"
```
/*
 * GET THE STATE OF A DISCRETE INPUT CHANNEL
 * 
 * Description: This function is used to get the current state of a discrete input channel. If the input
 * node is set to one of the edge detection modes, the number of edges detected is returned.
 * 
 * Arguments:
 *   n   is the discrete input channel. (n..DISK_INPUT_4)
 * 
 * Returns:
 *   0   if the discrete input is negated or, if no edge has been detected
 *   > 0 if edges have been detected
 *
*/

INT DISK_DOINT (unsigned n)
{
    INT val;

    if (n == DISK_INPUT_1)
    {
        val = INTDOWN(0, DISK_INPUT_1);
        return (val);
    }
    else
    {
        return (0);
    }

    /* Invalid channel */
```

Listing 8.1 (continued) DIO.C

```c
/* ***********************************************************************
 * DIO.H RC DIO.C
 * Description: This function is called to detect an edge low-going, high-going or both on the selected
 * discrete input.
 * Arguments: pDI is a pointer to the discrete input data structure.
 * Returns: none
 * ***********************************************************************
 */

static void DIO_Itrip (DIO_H *pDI)
{
    BOOLEAN trip;

    trip = FALSE;
    switch (pDI->edgeSelSel)
    {
    case DIO_EDGE_NEGEDGE: /* Negative-going edge */
        if (pDI->DIEdge == 0 && pDI->DIState == 0)
            trip = TRUE;
        break;
    case DIO_EDGE_POSEDGE: /* Positive-going edge */
        if (pDI->DIEdge == 0 && pDI->DIState == 1)
            trip = TRUE;
        break;
    case DIO_EDGE_BIUSEDGE: /* Both positive and negative-going */
        if ((pDI->DIEdge == 0 && pDI->DIState == 0) ||
            (pDI->DIEdge == 0 && pDI->DIState == 1))
            trip = TRUE;
        break;
    }

    if (trip == TRUE)
    {
        if (pDI->DIState == pDI->DIStatePrev)
        {
            /* No edge detected */
            return;
        }

        if (pDI->DIState == 0)
        {
            /* Yes, see used defined edge function */
            pDI->DIStatePrev = pDI->DIState;
        }
        else
        {
            /* Increment count of edges counted */
            pDI->DIStatePrev = 0;
        }

        pDI->DIState ^= pDI->DIStatePrev;
        /* Memorize previous input state */
    }
}
```

/*

UPDATE DECRETS IN CHANNELS

Description: This function processes all of the discrete input channels.
Arguments: None.
Returns: None.

*******************************************************************************/
Chapter 2: Discrete I/Os — 297

Listing 8.1 (continued)  DIO.C

```c
static void DIOISR(void)
{
    DIO_INT *dii;

    pdi = DIOINB(H);  // See if discrete input, enable is bypassed
    for (i = 0; i < DIO_INCOUNT; i++)
    {
        switch (dii->DIOIN[H][i])
        {
            case DIO_INPUT_HI:  // Input is forced high
                pdii->DIOIN[H][i] = 0;
                break;
            case DIO_INPUT_LO:  // Input is forced low
                pdii->DIOIN[H][i] = 1;
                break;
            case DIOḁ_INPUT_PASSTHR:  // Input is based on state of physical input
                pdii->DIOIN[H][i] = (DIOIN[H][i] == 1);
                break;
            case DIOḁ_INPUT_CO:  // Input is based on the complement state of input
                pdii->DIOIN[H][i] = !DIOIN[H][i] == 1;
                break;
        }
    }

    #if DIO_EDGE
    case DIO займ.EDGE_PIN:  // handle edge triggered mode
        DIOEDGEH(pdi);  // handle edge triggered mode
        break;
    #endif

    case DIO接入.PIN:  // DIO access to pin
        if (DIOIN[H][i] == 1)
        {
            pdii->DIOIN[H][i] = 0;
            pdii->DIOIN[H][i] = !DIOIN[H][i];
            break;
        }
        break;

    case DIO接入.PIN:  // DIO access to pin
        if (DIOIN[H][i] == 1)
        {
            pdii->DIOIN[H][i] = 1;
            pdii->DIOIN[H][i] = !DIOIN[H][i];
            break;
        }
        break;

    default:
        /* Handle other events */
        break;
```

/*
Listing 8.1 (continued)  DIO.C

```c
/*
 * Description: This function initializes the discrete I/O manager module.
 * Arguments: None
 * Returns: None.
 */

void DIOInit(void)
{
    int i;
    _DIO_IH_
    _DIO_DH_

    _DIO_IH_
    for (i = 0; i < _DIO_PORTS__; i++)
    {
        _DIO_PORT_IH_
        _DIO_PORT_DH_
    }
}
```
Listing 8.1 (continued)  DIO.C

```c
static void DIOWatch(void *data)
{
    data = data; /* Avoid compiler warning in CMS50 requirement */
    for (;;) {
        DIOWatch();  /* Delay between execution of DIO manager */
        DCON();     /* Read physical inputs and map to DCON channels */
        DUpdate();  /* Update all DCON channels */
        DSeq();     /* Map DCON channels to physical outputs */
    }
}

/* */

void DIEselpass (int n, int *val)
{
    if (n == DIO_NAMSEL0) {
        DIEselpass(); /* Set the state of the bypassed sensor. This function is used to simulate the presence */
        /* of the bypassed sensor. This function is only valid if the bypass 'switch' is open. */
        /* Arguments: n is the discrete input channel (0..DIO_NAMSEL-1). */
        /* val is the state of the bypassed sensor */
        /* 0 indicates a regular sensor */
        /* 1 indicates an asserted sensor */
        /* val is the number of edges detected in edge mode */
        /* Returns: None. */
    }
}

/* */
```

Listing 8.1 (continued)  DIO.C

/*! 
 * DESCRIPTION: This function is used to set the state of the sensor bypass switch. The sensor is
 * bypassed when the "switch" is open (i.e. DIO_DISABLE is set to false).
 * 
 * ARGUMENTS:  
 *  n: is the discrete input channel (0..DIO_MAX_INPUT-1) 
 *  state: is the state of the bypass switch
 *  DIO_DISABLE (i.e. the bypass "switch" is closed)
 *  DIO_ENABLE (i.e. the bypass "switch" is open)
 * 
 * RETURNS: None.
 */

void DIOSetBypass(int n, boolean state)
{
    if (n == DIO_MAX_INPUT) {
        OS_ENTER_CRITICAL();
        DIO_set_input(n, state);
        OS_EXIT_CRITICAL();
    }
}

"/*SIRN*/"
Listing 8.1 (continued) DTO.C

```c
/*
   DESCRIPTION: This function is used to configure the blink mode of the discrete output channel.
   
   Arguments:
   * state: The desired blink mode.
   * DTO_BLOB_ON: Blink is always enabled.
   * DTO_BLOB_ON_NORMAL: Blink depends on current request's state.
   * DTO_BLOB_ON_OFF: Blink depends on the complemented current request's state.
   * a: The number of 'blink' ON (1..255).
   * b: The number of 'blink' OFF (0..255).
   
   Returns: None.
*/

if (DTO_BLOB_MODE_SH && DTO_Option (DTO_BLOB_ON, DTO_BLOB_OFF, a, b, 0))
{
    DTO Đi "pdp"

    if (a < DTO_BLOB_ON { pDP = DTO_TRESHOLD; a = DTO_TRESHOLD; } /* Adjust threshold based on low often DTO runs */
        \ / DTO_BLOB_ON_NORMAL;
    pDP <= DTO_BLOB_ON NORMAL = node;
    pDP = a;
    pDP = pDP * DTO_BLOB_NORMAL;
    DTO_BLOB_ON_NORMAL="pdp"
    }"

    #undef "SPARCH"
```
Listing 8.1 (continued)  DIO.C

/** *
 * Description: This function is used to configure the mode of a discrete output channel.
 * Arguments:
 * mode: The desired mode and can be:
 * DIO_MODE_LO: Output is forced LOW
 * DIO_MODE_HI: Output is forced HIGH
 * DIO_MODE_GATE: Output is gated using state of DIO path
 * DIO_MODE_BLINKING: Output will be blinking synchronously with DIO path
 * low: Indicates whether the output will be inverted.
 * TRUE: force the output to be inverted
 * FALSE: does not change any inversion
 * Returns: None
 */

void DIOConfigure (DIOIHI n, DIOIHI mode, BOOLEAN inv)
{  
  if (n == DIOIHI_CE0) {
    OS_KP1.SetPort(n, mode);  
    OS_KP1.SetState(n, inv);  
  }  
}

/** *
 * Description: This function is used to obtain the status of the discrete output.
 * Arguments: n: is the discrete output channel (0..DIOIHI_CE0-1).
 * Returns: TRUE if the output is asserted.
 * FALSE if the output is negated.
 */

BOOLEAN DIOGet (DIOIHI n)
{  
  BOOLEAN out;

  if (n == DIOIHI_CE0) {
    OS_KP1.SetPort(n, mode);  
    OS_KP1.SetState(n, inv);  
    return out;
  } else {
    return TRUE;
  }
}

/** */
```c
Listing 8.1 (continued) ETO.C

/**
 * SEE IF BLOK IS ENABLED
 *
 * Description: See if block mode is enabled.
 * Arguments: pblk is a pointer to the discrete output data structure.
 * Returns: TRUE if blocking is enabled
 *          FALSE otherwise
 */

BOOL8 en;

en = FALSE;
switch (pblk->CTRLIN[0].pArg) {
  case TO блок.
    en = TRUE; /* Blink is always enabled */
    break;
  case TO блок. в блок.
    en = pblk->CTRLIN[0].pArg; /* Blink depends on user request's value */
    break;
  case TO блок. в блок.
    en = pblk->CTRLIN[0].pArg; /* Blink depends on the complemented user request's value */
    break;
}
return (en);
}
#endif
```
Listing 8.1 (continued) EXO.C

/*
 * SET THE STATE OF THE DIRECTED OUTPUT
 *
 * Description: This function is used to set the state of the discrete output.
 * Arguments: n is the discrete output channel (0..DIO_MAX-2-1).
 * state is the desired state of the output:
 * 0: RISE indicates a requested output.
 * 1: FALL indicates an asserted output.
 * Returns: None
 * Notes: The actual output will be complemented if 'rising' is set to TRUE.
 */
void DIOSetOutput (unsigned n, BOOL state) {
    if (n < DIO_MAX-2) {
        DIO[n].DIRECT = state;
        OL_SET (n, DIO[n].DIRECT);
    }
}

/* DIO_SET_INPUT */

/*
 * SET THE STATE OF THE Bypassed OUTPUT
 *
 * Description: This function is used to set the state of the bypassed output. This function is used to
 * override (or bypass) the application software and allow the output to be controlled
 * directly. This function is only valid if the bypass switch is open.
 * Arguments: n is the discrete output channel (0..DIO_MAX-2-1).
 * state is the desired state of the output:
 * 0: RISE indicates a requested output.
 * 1: FALL indicates an asserted output.
 * Returns: None
 * Notes: 1) The actual output will be complemented if 'rising' is set to TRUE.
 *        2) In blank mode, this allows blocking to be enabled or not.
 */
void DIOSetBypass (unsigned n, BOOL state) {
    if (n < DIO_MAX-2) {
        if (DIO[n].DIRECT == 0) {
            DIO[n].DIRECT = state;
            OL_SET (n, DIO[n].DIRECT);
        }
    }
}

/* DIO_SET_INPUT */
Listing 8.1 (continued) DIO.C

/*
 * Description : This function is used to set the state of the output bypass switch. The output is bypassed when the switch is open (i.e. XOR mode) is set to TRUE.
 * Arguments : n is the output which will be bypassed (0, 2 or 3).
 *             state is the state of the bypass switch (TRUE or FALSE).
 *             True enables output bypass (i.e. the switch is closed)
 *             False disables output bypass (i.e. the switch is open)
 * Returns : None.
 */

void setBypass(int n, BOOLEAN state)
{
    if (n < XOR_MODE) {
        setBypassControl();
        OXR[n] = (XORstate = state);
    }
}

/*
 */

/*
 * Description : This function is used to set the maximum value taken by the asynchronous counter which is used in the asynchronous clock mode.
 * Arguments : val is the maximum value for the counter (1...255)
 * Returns : None.
 */

void setMaxValue(int val)
{
    if (asyncMode != TRUE) {
        asyncCounter = val;
        OXR[0] = (XORstate = TRUE);
    }
}

/*
*/

Chapter 8: Discrete I/Os — 305
Listing 8.1 (continued)  DIO.C

/*
  *********************************************************
  DIOCIOC1OUPY OPTIONS
  *
  DESCRIPTION: This function is called to process all of the discrete output channels.
  * Arguments:  none.
  * Returns:   none.
  *********************************************************
*/
Listing 8.1  (continued)  DIO.C

static void DIOHandler (void)
{
  int i;
  short pno;

  pno = 40007; //
  for (i = 0; i < CNT_MAX_DIO; i++) { /* Process all discrete output channels */
    if (pno-- >= 40007) /* if DIO channel is enabled */
      pno = 40007;
    else { /* Obtain control state from application */
      out = PAGE1;
      if (pno-- >= 40007) { /* Assume that output will be low until changed */
        case DIO_MODE_LOW:
          /* Output will be low */
          break;
        case DIO_MODE_HIGH:
          /* Output will be high */
          out = PAGE0;
          break;
        case DIO_MODE_TOGGLE:
          /* Output is toggled on state of user supplied state */
          out = pno-- >= 40007;
          break;
        if (pno-- >= 40007) /* Sync. Blink mode */
          /* See if Blink is enabled ... */
          if (pno-- >= 40007) /* ... yes, High when below threshold */
            out = PAGE0;
          break;
        case 0x0000:
          /* Async. Blink mode */
          /* See if Blink is enabled ... */
          if (pno-- >= 40007) /* ... yes, High when below threshold */
            out = PAGE0;
          break;
      }
    }
  }
  if (pno-- >= 40007) /* See if output needs to be inverted ... */
    if (pno-- >= 40007) /* ... yes, complement output */
      if (pno-- >= 40007) /* ... no, no inversion */
        pno--;
    break;
  if (pno-- >= 40007) /* Want to next DIO channel */
    if (DIO_mode < DIO_MODE_DISABLE &&
        DIOmodeCtrl) { /* Update the synchronous free running counter */
      DIOmodeCtrl = 0;
    } else { /* DIOmodeCtrl = 0; */
  }
}
#endif
Listing 8.1 (continued) DIO.C

```c
#define CMUC

******************************************************************************

* DESCRIPTION: This function is by DIOInit(). to initialise the physical I/O used by the DIO driver.
* ARGUMENTS: I None.
* RETURN: I None.

******************************************************************************

void DIOInitIO (void)
{
    /* Setup CMUW, CMU2; */
    outp(CMUW, CMU2);

    /* I Port A = CMU, Port B = IN, Port C = CMU */
    ...
}

******************************************************************************

* DESCRIPTION: This function is called to read and map all of the physical inputs used for discrete inputs and map these inputs to their appropriate discrete input data structure.
* ARGUMENTS: I None.
* RETURN: I None.

******************************************************************************

void DIOI (void)
{
    DIOUI[0];
    DIOUI[1];
    DIOUI[2];
    DIOUI[3];

    /* Map all 64 bits to first 8 IO channels */
    in = ...;
    for (i = 0; i < 8; i++)
    {
        DIOUI[0] = DIOUI[0] & (1 << i);
        in = in | (1 << i);
    }

    /* Read the appropriate port */
    DIOUI = DIOUI & (1 << i);
}
```

Listing 8.1 (continued) DIO.C

/*
 * Description: This function is called to map all of the discrete output channels to their appropriate
 * physical destination.
 * Arguments: " "
 * Returns: None.
 */

void COMA (void)
{
    DIO.DO *pdo;
    DNUM i;
    DNUM done;
    DNUM nks;

    pdo = MSTEA(2);  /* Point at first discrete output channel */
    FDK = DDK();   /* First DO will be mapped to bit 0 */
    for (i = 0; i < 8; i++) { /* Load 8 bit port image */
        if (pdo->COM0 = TRUE) { /* Map first 8 DO to 8 bit port jDMI */
            out = nks;
            PDO++;
        }
    }
    out = nks;  /* Output port image to physical port */
}

#define
Listing 8.2  DIO.H

ﾉ/

ﾉ/  Embedded Systems Building Blocks
ﾉ/  Complete and Ready-to-Use Modules in C
ﾉ/  Discrete I/O Module
ﾉ/  *
ﾉ/  * Filename : DIO.H
ﾉ/  * Programmer : Jean-J. Laborie
ﾉ/  * All Rights Reserved
ﾉ/  *
ﾉ/  */
ﾉ/  */
ﾉ/  */  CONVENTION Comments
ﾉ/  */
ﾉ/  */
ﾉ/  #ifndef CMLK
ﾉ/  #define DIO_MAX_PROG 40
ﾉ/  #define DIO_MAX_DMA_PROG 1
ﾉ/  #define DIO_MAX_DMA_DMA 512
ﾉ/  #define DIO_MAX_DIO 8 /* Maximum number of Discrete Input Channels (1..255) */
ﾉ/  #define DIO_MAX_DO 8 /* Maximum number of Discrete Output Channels (1..255) */
ﾉ/  #define IN_EDGE_TRIG 1 /* Enable code generation to support edge trip when 1 */
ﾉ/  #define EN_DISABLE_EDGE_TRIG 1 /* Enable code generation to support blink mode when 1 */
ﾉ/  #endif

ﾉ/  #ifdef DIO_CLAIMERS
ﾉ/  #endif DIO_CLAIMERS
ﾉ/  #ifdef DIO_REAL
ﾉ/  #endif DIO_REAL
ﾉ/  #endif
# Listing 8.2 (continued) DIO.H

```c
/*
 * DISCRETE INPUT CONSTANTS
 */

#define DE_MODE_LOW 1 /* Input is forced low */
#define DE_MODE_HIGH 1 /* Input is forced high */
#define DE_MODE_ABSENT 2 /* Input is based on state of physical input */
#define DE_MODE_LOW 3 /* Input is based on complement of the physical input */
#define DE_MODE Edge_LOW_FALLING 4 /* Low going edge detection of input */
#define DE_MODE Edge_HIGH_FALLING 5 /* High going edge detection of input */
#define DE_MODE Edge_LOW_RISING 6 /* Both low and high going edge detection of input */
#define DE_MODE Edge_LOW_FALLING 7 /* Low going edge detection of input */
#define DE_MODE Edge_HIGH_FALLING 8 /* High going edge detection of input */

#define Edge_FALLING 0 /* Negative going edge */
#define Edge_RISING 1 /* Positive going edge */
#define Edge_BOTH 2 /* Both positive and negative going */

/*#ifndef*/
```

---

**Chapter 8: Discrete I/O**

311
Listing 5.2 (continued) DIO.H

/  
/意识到输出配置
/  
/*

#define DIO_MODE_LOW 0 /* Output will be low */
#define DIO_MODE_HIGH 1 /* Output will be high */
#define DIO_MODE_DEFAULT 2 /* Output is based on state of user supplied state */
#define DIO_MODE_BLINK_HIGH 3 /* Blink high mode */
#define DIO_MODE_BLINK_DEFAULT 4 /* Blinking high mode */

#define DIO_MODE_EN 4 /* Blink is always enabled */
#define DIO_MODE_BLINK_DISABLE 1 /* Blink depends on user request's state */
#define DIO_MODE_BLINK_ENABLE 2 /* Blink depends on the complemented user request's state */
/*

/ DATA TYPES
/  

/typedef struct dio_d { /* DESCRIBE INPUT CHANNEL DATA STRUCTURES */
int DIO0; /* Current state of sensor input */
int DI0; /* State of discrete input channel (or of transition) */
int DIO2; /* Previous state of DIO for edge detection */
int DIO3; /* Disable edge detection */
int INPUT; /* Discrete input channel mode selector */
#endif

void (unsigned int void (*)), /* Function to associate if edge triggered */
args passed to function when edge detected */
#endif

/typedef struct dio_o { /* DESCRIBE OUTPUT CHANNEL DATA STRUCTURES */
int DIO0; /* Current state of discrete output channel */
int DIO1; /* Discrete output channel mode selector */
int DIO2; /* Bypass enable switch (bypass when TRUE) */
int DIO3; /* Bypass output channel mode selector */
int INPUT; /* Discrete output channel mode selector (input when TRUE) */
#endif

#define DIO_MODE_INPUT 1 /* Blinks via CH3 */
#define DIO_MODE/logout /* Asynchronous blink mode period */
#define DIO_MODE_BLINK /* Asynchronous blink mode period counter */
#endif

/*DIO.H*/
Listing 8.2 (continued) DIO.H

/*
 **************************************************************************
 * ** GLOBAL VARIABLES **
 **************************************************************************
 */

void DIOInit(void);

void DIOpinmode(DINO n, DINO node);
DINO DIOget(DINO n);
void DIOgetstate(DINO n, BOOLVAL vstate);
void DIOclearstate(DINO n, DINO val);

#if defined(UNIX)
void DIOset(DINO n);
void DIOCLEAR((DINO n), DINO v);
#endif

void DIOpinmode(DINO n, DINO node, BOOLVAL inv);
BOOLVAL DIOgetstate(DINO n, DINO node, BOOLEAN state);
void DIOsetstate(DINO n, DINO node, BOOLEAN state);

#if defined(UNIX)
void DIOCLEAR((DINO n), DINO node, DINO v);
void DIOCLEAR((DINO n), DINO v);
#endif

/*
 * FUNCTION PROTOTYPES
 **************************************************************************/

void DIOuninit(void);
void DIOinit(void);
void DIONode(void);

Chapter 8: Discrete I/Os — 313
Chapter 9

Fixed-Point Math

Most low-end microprocessors (typical of embedded processors) do not provide hardware-assisted floating-point math. Microprocessor manufacturers unfortunately seem to feel that floating-point math is not very important in embedded systems. This has not been my experience. Fortunately, ANSI C compilers allow you to use floating-point math but at a cost: floating-point libraries require extra ROM and RAM but most importantly, they require more processing time than integer math. For example, floating-point addition could take hundreds of microseconds on a low-end, 8-bit microprocessor, whereas it typically takes only a few microseconds to perform a 16-bit integer addition. Multiplications and especially divisions are even worse. As an embedded system programmer, you are often confronted with the task of writing the fastest and smallest possible code for real-time operations. This chapter will show you how to perform basic arithmetic operations on fractional numbers by using only integers. In other words, this chapter will answer the question: "Without using floating-point arithmetic, how would you add 12.34 and 987.654, multiply 3.1416 by 5.4, or divide 0.0056 by 98.7?"

Throughout this chapter, I will be using 16-bit integers, but most of the concepts presented here apply to any integer size. This chapter will show you how to use the concept of fixed-point math to get the most out of integer arithmetic. Chapter 10 will make use of the information presented in this chapter.

9.00 Fixed-Point Numbers

Fixed-point is an alternative form for expressing numerical values. Fixed-point math is integer math, but because it allows fractions, it is much more versatile and often can substitute for slower and more cumbersome floating-point operations. The idea of fixed-point math is to trick the computer into thinking you are talking about an integer when in fact you, the programmer, know that you are dealing with a number that has a fractional component.

Figure 9.1a shows a 16-bit integer. The computer thinks only in bits. In integer arithmetic, the bit positions are said to represent 2 to progressively higher powers starting from the right. The bit string 0000000000001000, therefore, represents the number 16.
Figure 9.1.a Signed and unsigned 16-bit integers.

\[
\begin{array}{c|c}
\text{Unsigned 16-bit integer} & 0 \times 2^{15} = 0 \\
0.0.0.0.0.0.0.0.0.0.1.1.0.0.0.0 & 0 \times 2^{15} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 16 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{Signed 16-bit integer} & 0.0.0.0.0.0.0.1.0.0.0.0.0.0.0.0.0 \\
0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0 & 0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0 \\
& \text{Sum} = 16 \\
\end{array}
\]

A practitioner of fixed-point math would observe that there is an implied decimal point (called a radix point) to the right of the rightmost bit position and would ask, "Why must it fall there? Why can't I put the radix point somewhere else?" In other words, why must the rightmost bit represent \(2^{-15}\)?

Figure 9.1.b shows the same 16-bit string. In this case, the programmer decides to place the radix point between the 5th and 6th bit positions, which make the rightmost bit \(2^{-5}\). The string 0000000000010000 is now set 16, but 0.5. Another way to look at this is to say that the integer 16 has been scaled by \(2^{5}\) (unspecified by \(2^{1}\), or 0.03125).

\[
16 \times 2^{-3} = 0.5
\]

Figure 9.1.b Signed and unsigned fixed-point numbers with radix point between 5th and 6th bits.

\[
\begin{array}{c|c}
\text{Unsigned 16-bit integer} & 0 \times 2^{15} = 0 \\
0.0.0.0.0.0.0.0.0.0.0.0.0.0.1.0.0 & 0 \times 2^{15} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& 0 \times 2^{1} = 0 \\
& \text{Sum} = 0.5 \\
\end{array}
\]

Decimal Point
(Radix point)
(Defined by Programmer)
Chapter 9: Fixed-Point Mark — 317

The computer, then, thinks it is working with the integer 16, but the programmer independently maintains a record of how the 16 should be scaled.

By manipulating the position of the radix point, a programmer can scale integers into fractional values. The location of the radix point defines a convention for how the program will interpret a 16-bit string. As the radix point moves to the left (increasing the fractional portion of the string) the fraction becomes more precise and the overall range of the number diminishes (because there are fewer whole-number places).

The unsigned integer of Figure 9.1 b can be used to represent numbers having a range of 0.0 to 2047.96775, while the signed integer can represent numbers between −1024.0 to 1023.96875 (assuming two's complement). Both signed and unsigned numbers have a resolution of 1/32nd (0.03125). You can use fixed-point to represent distances, surfaces, volumes, temperatures, pressures, etc. Depending on the application, you can fix the position of the radix point elsewhere to suit the range of numbers you have to deal with.

Figure 9.2 shows how you can represent temperatures from −459.67 °F (0 °Kelvin, absolute 0) to +2048 °F by using an 11-bit integer and a 4-bit fraction. An integer value of 11528 represents a temperature of 720.5 °F (11.28 x 2^-3). Using this format, temperatures can be represented with a 1/16th °F resolution. The temperature scale is an ideal use for fixed-point math because the range is well defined, so the programmer can easily set the location of the radix point in advance.

Figure 9.2 Representing temperatures from −459.67 °F to 2047 °F.

Signed 16-bit integer

11-bit Integer

4-bit Fraction

Decimal Point

(Radix point)

When your program performs arithmetic operations (add, subtract, multiply, or divide) on fixed-point numbers, it actually manipulates integers. (Microprocessors do not provide mechanisms to represent fixed-point numbers.) This means that the programmer must personally keep track of the position of the radix points. To represent fixed-point numbers, I will use the following notation:

Fixed-point number = mantissa x 2^exponent,

where $S$ means that the mantissa needs to be scaled by $2^{−S}$ to determine the value of the fixed point number. The exponent is sometimes called the scale factor. The mantissa is always an integer number. I use this notation to differentiate the fixed-point notation from the floating-point notation mantissa x $2^{exponent}$. Following are some examples of the use of this notation.

$S$S-3 represents 0.625 or $5 \times 2^{-3}$ or $5\times 8$

31S-8 represents 0.121 or $31 \times 2^{-8}$ or $31 \times 256$

−123S-16 represents $-0.001877$ or $-123 \times 2^{-16}$ or $-123 \times 65536$

The mantissa is shown in bold to emphasize that the fixed-point number is actually represented using an integer whereas the exponent is maintained mentally by the programmer.
Scaling is done to allow almost any number to be represented using a 16-bit integer. The position of the radix point is determined from the largest number that you need to represent. Equation (9.1) shows how to obtain the mantissa and the exponent (scale factor) for any positive value \( x \) between 0.0 and 65535.0.

**POSITIVE NUMBERS \( 0.0 < x \leq 65535.0 \):**

\[
\begin{align*}
\text{[9.1]} & \quad \text{factor} = -\text{INT} \left( \frac{\log(65535)}{\log(2)} \right) \\
\text{mantissa} &= \text{INT} \left( 2^{-\text{factor}} \times x + 0.5 \right)
\end{align*}
\]

where \(\text{INT}()\) means that you take the integer portion of the result. In other words, the result is truncated. \(\log()\) is the logarithm of the number in parentheses (either \(\log_{10}()\) or \(\log_{2}()\)). When \(x = 0.0\) both the mantissa and the factor are 0. To represent the number 1.2345 using the fixed-point number notation, you would substitute 1.2345 into Equation (9.1) as follows:

\[
-15 = -\text{INT} \left( \frac{\log(1.2345)}{\log(2)} \right)
\]

\[
40452 = \text{INT} \left( 2^{15} \times 1.2345 + 0.5 \right)
\]

Thus, the number 1.2345 is written as \(40452.15\).

Equation (9.2) shows how to obtain the mantissa and the exponent for a positive value of \( x \) that is greater than 65535.0.

**POSITIVE NUMBERS \( x > 65535.0 \):**

\[
\begin{align*}
\text{[9.2]} & \quad \text{factor} = \text{INT} \left( \frac{\log(x)}{\log(2)} \right) + 1 \\
\text{mantissa} &= \frac{x}{2^{\text{factor}}}
\end{align*}
\]

Again, \(\text{INT}()\) means that we take the integer portion of the result. \(\log()\) is the logarithm of the number in parentheses. For example, the number 107573 is represented as:

\[
1 = \text{INT} \left( \frac{\log(107573)}{\log(2)} \right) + 1
\]
Thus, the number 107573 is written as $53786.1$. Note that in this case, we lose resolution because we would actually need 17 bits to represent 107573 but we only have 16 bits.

Equation (9.3) shows how to obtain the mantissa and the exponent for any signed value $x$ between $-32767.0$ to $+32767.0$ (inclusively).

**SIGNED NUMBERS** $(-32767.0 \leq x \leq +32767.0$, except 0.0):

\[
\text{factor} = \text{INT}\left( \frac{\log \frac{32767}{|x|}}{\log(2)} \right)
\]

\[
\text{mantissa} = 2^{\text{factor}} \times x
\]

where \(\text{INT}\) means that we take the integer portion of the result. In other words, the result is truncated.

\(\text{INT}\) means the absolute value of the number to scale. \(\log(\cdot)\) is the logarithm of the number in parentheses. When $x$ is 0.0, both the mantissa and the factor are 0.

Equation (9.4) shows how to obtain the mantissa and the exponent for a signed integer that is less than $-32767.0$ and greater than $+32767.0$.

**SIGNED NUMBERS** $(-32767.0 > x > +32767)$

\[
\text{factor} = \text{INT}\left( \frac{\log |32767|}{\log(2)} \right) + 1
\]

\[
\text{mantissa} = \frac{x}{2^{\text{factor}}}
\]

Again, \(\text{INT}\) means that we take the integer portion of the result. \(\text{INT}\) is the absolute value of the number to scale, and \(\log(\cdot)\) is the logarithm of the number in parentheses.

### 9.01 Fixed-Point Addition and Subtraction

To add or subtract two fixed-point numbers, the exponent of both numbers must be the same. For example, you could not add the signed fixed-point number $204805.15$ (0.6250) with $317455.18$ (0.1211) because they do not represent the same order of magnitude. In order to add these numbers, you would first convert the smaller number (317455.18) to the order of magnitude of the larger number. You would do this by adding 3 to the exponent (which is the same as multiplying by $2^3$, or 8) and then dividing the mantissa by 8. The number would be $39685.15$ (i.e., $39685/32768$). The result of the addition is thus
230 — Embedded Systems Building Blocks, Second Edition

24405s−15 (0.746094). Pretty simple, right? Actually, things get a little trickier when you add two numbers and the result exceeds unity. For example:

0.99 + 0.99 = 1.98 or

234405s−15 + 234405s−15 = 648805s−15

What actually happens here is that the addition overflows because the maximum value for a signed 16-bit fixed-point number can only be 32767! In this case, you can avoid the overflow by scaling both numbers to 5−14 instead of 5−15 as shown following this paragraph. You will thus need to be careful when you add or subtract two fixed-point numbers.

0.99 + 0.99 = 1.98 or

162205s−14 + 162205s−14 = 324405s−14

9.02 Fixed-Point Multiplication

To multiply fixed-point numbers, you simply multiply the mantissa of the two numbers and add the exponents. For example, we can multiply the two signed 16-bit fixed-point numbers:

0.6250 × 0.1211 = 0.075608 or

204805s−15 × 317455s−18 = 65013760005s−73

One thing to note here is that when you multiply two signed 16-bit numbers, the result is a 30-bit number. Because of this, your C compiler needs to support signed longs (32-bit numbers). In the previous example, you must divide the number by 327685s−15 (i.e., this is a division by 1.0 and does not change the result) to obtain a signed 16-bit result. A division by 327685s−15 simply involves shifting the mantissa right 15 places. In this case, the result would be 198405s−18 (or 0.075684).

For unsigned fixed-point numbers, the multiplication yields a 32-bit result. For example, 0.6250 × 0.1211 looks like this:

409605s−16 × 634915s−19 = 260005136005s−35

A division of 65536 would make the previous result fit back into an unsigned 16-bit integer: 396815−19 (or 0.075660). Note that the result is more accurate than its signed version because more bits were used in the unsigned multiplication.

9.03 Fixed-Point Division

Divisions are always trickier (and slower) than multiplications. For example, instead of dividing a number by 10, you should consider multiplying the number by 0.1 (or 262145−18, signed). If you have to perform a division, however, you simply divide the mantissas and subtract the exponents as:

0.2545 +−10.987 = −0.027543 or

307305s−17 + −225015s−11 = −15·6 (−0.015625)
Note how the result is totally incorrect. This is because the division produced a result of \(-1\) and a remainder of 8335. C compilers don’t know what to do with remainders. To avoid this problem, you simply need to scale the dividend by \(32768^{15}\) and remember that the final result has been multiplied by \(32768\):

\[
(307665 \times 17 \times 32768^{15}) \div -225915^{15} = -447605^{15} \div (2^{16} \div 0.0021343)
\]

Note that the mantissa of the result doesn’t fit in a 16-bit signed number. Because of this, the result needs to be adjusted as follows:

\[
-447605^{15} \div -25 -1 = -233805^{15} \div (2^{16} \div 0.0021343)
\]

The overflow problem will occur whenever the mantissa of the numerator is greater than the mantissa of the denominator. You code will have to check for this situation.

### 9.04 Fixed-Point Comparison

Comparing two fixed-point numbers presents a problem similar to the problem of adding and subtracting: the exponent of both numbers must be the same. For example, comparing \(204805^{15}\) with \(337455^{15}\) requires that you adjust the smaller of the two numbers to match the scale of the larger. \(337455^{18}\) would thus become \(39685^{15}\) (i.e., \(39685 / 2768\)). Once both numbers represent the same order of magnitude, comparing the two numbers is simply a matter of comparing the mantissas.

### 9.05 Using Fixed-Point Arithmetic, Example #1

Suppose you needed to compute the circumference of a circle that can vary in diameter from 1.22 to 20.8 inches. The circumference of a circle is given by:

\[
\text{Circumference} = \pi \times \text{Diameter}
\]

Because diameters are positive quantities, we will use unsigned fixed-point numbers. \(\pi\) can be represented as \(514725^{14}\) (actually \(3.141602\)). As shown in Figure 9.3, we need a 5-bit integer to represent the diameter of the circle; the other 11 bits of an unsigned 16-bit integer number are used to hold the fraction. In other words, the diameter will be scaled by \(2^{11}\). Numbers for the diameter will be represented as \(<\text{mantissa}>5^{11}\).
Figure 9.3 Fixed-point representation for circle diameter.

```
<table>
<thead>
<tr>
<th>5 bit Integer</th>
<th>11 bit Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 bit Integer</td>
<td></td>
</tr>
</tbody>
</table>
```

The circumference of the circle is computed in C as follows:

```c
#include <stdint.h>

struct {
    uint16_t x;

    x = (uint16_t)((3.14159265358979323846 * (int32_t) diameter) >> 16);
    return (x);
}
```

Multiplying two 16-bit unsigned integers will yield a 32-bit result, so you must adjust the resultant mantissa by dividing by $\#5336$ (i.e., shifting right 16 places). The exponent of the result is determined as follows. $x$ has an exponent of $-14$ and the diameter has an exponent of $-11$. However, the right shift is the same as dividing by $65536^{16}$ and thus, the exponent of the result is $(−14) + (−11) = (−25)$.

Our minimum circumference is obtained by substituting a 1.22 (29985–11) inch diameter circle in the previous code. The multiplication yields 3205770665–25. After the shift, the result is 19615–9 (3.830078) which is within about 0.07 percent of the correct result of 3.832743. Our maximum circumference is obtained by substituting a 20.8 (425985–11) inch diameter circle in the previous code. The multiplication yields 21920642565–25. After the shift, the result is 334565–9 (65.343750) which is within about 0.002 percent of the correct result of 65.345127.

9.06 Using Fixed-Point Arithmetic, Example #2

Computing the volume of a cylinder involves more multiplications. The formula for the volume $V$ of a cylinder is:

\[
V = \frac{\pi \times (Diameter)^2 \times Length}{4}
\]

Suppose the cylinder length varies from 9 to 24 inches, and the diameter varies from 1 to 12 inches. To compute the volume of a cylinder, I will again use unsigned integer math because all arguments are strictly positive. $\pi$ can be represented as 3.14159265358979323846 (actually 3.141600). To represent the length of the cylinder, we need 5 bits for the integer portion (up to 31 inches). The other 11 bits of an unsigned 16-bit integer number are used to hold the fraction; in other words, the length will be scaled by $2^{11}$. Similarly, the diameter will require 4 bits for the integer portion and 12 bits for the fraction. This is shown in Fig.
use 9.4. Numbers representing the length will be represented as mantissa-S-11 while numbers for the
diameter will be represented as mantissa-S-12.

**Figure 9.4** Fixed-point representation for cylinder length and
diameter.

```
<table>
<thead>
<tr>
<th>Unsigned 16-bit integer</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 bit Integer</td>
<td></td>
</tr>
<tr>
<td>11 bit Fraction</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Unsigned 16-bit integer</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bit Int</td>
<td></td>
</tr>
<tr>
<td>12 bit Fraction</td>
<td></td>
</tr>
</tbody>
</table>
```

The volume of the cylinder is computed in C as follows:

```c
INT64 Volume(INT64 length, INT64 diameter)
{
    INT64 vol;
    INT64 dia;

    dia = (INT32) (diameter);
    x = (31472 * dia) >> 16; /* 3-10 Result */
    x = (x * dia) >> 16; /* 8-6 Result */
    x = (x * (INT24) length) >> 16; /* 8-1 Result */
    return ((INT64)(x)); /* 8-3 Result */
}
```

Each multiplication is carried out separately because you must convert the resulting 32-bit mantissa
to a 16-bit mantissa. The exponent of the result is S-10 (S-14 X S-12 + S-16). The diameter is multi-
plied by the intermediate result and again, the new result is adjusted. The exponent of this new result is
S-6 (S-10 X 10 + S-12 + S-16). Finally, the length is multiplied by the surface of the circle to obtain the
volume. The exponent of the result is S-1 (S-6 X S-11 + S-16), however, you can avoid dividing by 4
simply by changing the scale of the result. Thus, the final exponent is S-3.

Our minimum volume is obtained by substituting a 9-inch long (18432-11) 1-inch diameter cylin-
der (4096-12).

1st Multiplication: $51472 \times 4096-12 = 2108288$ or $32178 \times 4096-12$ after the shift.
2nd Multiplication: $5275-10 \times 4096-12 = 1317683252$ or $2015-6 \times 4096-12$ after the shift.
3rd Multiplication: $2015-6 \times 18432-11 = 3704852517$ or $565-1$ after the shift.
The returned value is actually scaled S-3 and thus, the final result is 565-3 (or 7.00). The real volume should be 7.06858, which results in an error of 0.98 percent. Performing the same operations using our maximum values (12-inch diameter (491525-12) and a 24-inch length (491525-11)) will yield the following results:

1st Multiplication \[514725 \times 491525-12 = 25299517445-26\] or 386045-10 after the shift.

2nd Multiplication \[386045-10 \times 491525-12 = 189746380085-22\] or 289535-6 after the shift.

3rd Multiplication \[289535-6 \times 491525-11 = 14230978565-17\] or 217145-1 after the shift.

The returned value is then 217145-3 (2714.25). The actual volume is 2714.336 yielding an error of only 0.003 percent. One thing to note is that the second multiplication produced a number that is less than half of the full scale. In other words, 28953 is less than half the full range of an unsigned 16-bit number (0 to 65535). By shifting left by 15 places instead of 16 places, you would actually obtain better accuracy from that point on, as shown:

1st Multiplication \[514725 \times 491525-12 = 25299517445-26\] or 380405-10 after the shift.

2nd Multiplication \[380405-10 \times 491525-12 = 189746380085-22\] or 579065-7 after a shift of only 15 places.

3rd Multiplication \[579065-7 \times 491525-11 = 28461957125-18\] or 434295-4 after the shift.

The returned value is this case is 434295-4, which is 274.3125, but the computation was performed with better accuracy throughout. This improvement is accuracy would help when computing smaller volumes. The final code would be:

```c
#define (INF160) Volume(INF160 length, INF160 diameter)
{
    INF32U x;
    INF32U dia;

dia = (INF32U)diameter;
    x = (INF32U) length * dia; /* S-2 Result */
    x = (x * dia) >> 16; /* S- 7 Result */
    x = (x * (INF32U) length) >> 16; /* S- 2 Result */
    return ((INF160)x); /* S- 4 Result */
}
```
9.07 Using Fixed-Point Arithmetic, Example #3

You can use fixed-point arithmetic to convert °C (degrees Celsius) to °F (degrees Fahrenheit). The equation for converting °F to °C is:

\[ ^\circ C = \left( ^\circ F - 32 \right) \times \frac{5}{9} \]

In order to determine how to implement the conversion equation using fixed-point arithmetic, you need to know the range of temperatures that you will be dealing with. Suppose that you are interested in temperatures from -40 °F to 250 °F. The range chosen forces you to use signed integer arithmetic. Also, you need 8 bits to represent temperatures up to 250 °F, and thus, 7 bits will be used to represent fractional degrees. The bias of 32 °F is represented as 40965-7, while the constant multiplier 5/9 can be represented as 182045-15. The code to perform the conversion is:

```
// Fixed-to-Fixed conversion from °F to °C

int16_t ConvertTemp(int16_t temp) {
    return ((int32_t)(temp - 4096) * 18204) >> 15;
}  // Result is S-7 */
```

The temperature in °C is scaled S-7 (i.e., S-7 X S-15 + S-15). Performing the conversion from °C to °F is just as simple. The equation is:

\[ ^\circ F = \frac{^\circ C \times 9}{5} + 32 \]

Again, the 32 °F constant is 40965-7, while the constant multiplier 9/5 is 24915-14. The conversion code is:

```
// Fixed-to-Fixed conversion from °C to °F

int16_t ConvertTemp2(int16_t temp) {
    x = ((int32_t)(temp * 2491)) >> 14;
    return (x + 40965) >> 8;  // Result is S-7 */
}
```

Note that to obtain an S-7 result, I had to divide the result of the multiplication by 16384 instead of 32768.
9.08 Conclusion

To use fixed-point arithmetic, you need to know the range of values that the variables can take. Fixed-point arithmetic operations will generally execute quickly because most microprocessors are good at performing integer operations. This performance is at the expense of accuracy and complexity. To improve the accuracy you have to use more bits. Using fixed-point arithmetic produces large errors when using small numbers (i.e., numbers at the bottom of the scale) and decent results using large numbers. For large numbers, the improvement in accuracy is at the cost of using more bits. Fixed-point works very well when the dynamic range of the numbers is small.

9.09 Bibliography

Crawford, Charles
"Floating-Point Arithmetic with the TMS32010"
Houston, TX
Texas Instruments Inc., 1986

Institute of Electrical and Electronics Engineers, Inc.
ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic
345 East 47th Street
New York, NY 10017

Knuth, Donald E.
Reading, Massachusetts
Addison-Wesley Publishing Company
ISBN 0-201-03822-6

Morgan, Don
Numerical Methods, Real-Time and Embedded Systems Programming
San Mateo, CA
M&T Publishing, Inc.

Pomise, Jeff
"Questions & Answers"
Microsoft Systems Journal
March 1993, p85,86

Simar, Ray Jr.
"Floating-Point Arithmetic with the TMS32010"
Houston, TX
Texas Instruments Inc., 1986
Chapter 10

Analog I/Os

Natural parameters such as temperature, pressure, displacement, altitude, humidity, flow, etc., are analog. In other words, the values taken by these parameters can change continuously instead of in discrete steps. To be manipulated by a computer, these analog parameters must be converted to digital. This is called analog-to-digital conversion.

Certain analog parameters can also be controlled. For example, the speed of an automobile is adjusted by changing the position of the throttle. The exact position of the throttle depends on many factors, such as wind resistance, whether you are going uphill or downhill, etc. You can control the flow of liquids or gases by adjusting the opening of a valve. (Flow, in this case, is not necessarily proportional to the opening of the valve, but this is a different issue.) The position of the heads in some hard disk drives is controlled by voice coil type actuators. An actuator is a device that converts electrical or pneumatic signals into linear motion. To be controlled by a computer, analog parameters must be converted from their digital form to analog. This is called digital-to-analog conversion.

This chapter discusses software issues relating to analog-to-digital conversions and digital-to-analog conversions. I will also describe how I implemented an analog I/O module. The analog I/O module offers the following features:

- Reads and scales from 1 to 250 analog inputs.
- Updates and scales from 1 to 250 analog outputs.
- Each analog I/O channel can define its own scaling function.
- Your application obtains Engineering Units from analog input channels instead of ADC counts.
- Your application provides Engineering Units to analog output channels instead of DAC counts.

This chapter assumes you understand the concept of fixed-point math, described in Chapter 9.
10.00 Analog Inputs

A typical analog-to-digital system generally consists of the following circuit elements:

- transducer
- amplifier
- filter
- multiplexer
- analog-to-digital converter (ADC)

The interconnection of these components is shown in Figure 10.1. The inputs to the system are the physical parameters to measure (pressure, temperature, flow, position, etc.).

**Figure 10.1 Analog-to-digital conversion.**

The physical parameter is first converted into an electrical signal by a transducer. Transducers are available to convert temperature, pressure, humidity, position, etc., to electrical signals. An amplifier is generally used to increase the amplitude of the transducer output to a more usable level for further processing (typically between 1 and 10 volts); the output of a transducer may produce a signal in the millivolt to millivolt range. The amplifier is frequently followed by a low pass filter, which is used to reduce unwanted high-frequency electrical noise. The process described previously is usually called input conditioning and each conditioned input is also referred to as an analog input channel. Analog input channels are multiplexed into an analog-to-digital converter (ADC) because ADCs are often expensive devices. The ADC converts each analog input signal to digital form. The microprocessor is responsible for selecting which analog input it wants to convert and also for initiating the conversion process for the selected channel. The block diagram of Figure 10.1 can be augmented by adding a sample-and-hold stage between the multiplexer and the ADC which would be used to ensure that the level of the signal is constant while a conversion is taking place.

The process of converting analog signals to digital is a complex topic and is covered in great details in many books. In this book, I will concentrate mostly on some of the software aspects. Analog-to-digital conversion basically consists of transforming a continuous analog signal into a set of digital codes. This is called quantizing. Figure 10.2 shows how a 0-to-10 volt signal is quantized into a 3-bit code.
Figure 10.2 Quantizing an analog signal.

There are several important points to note about Figure 10.2. First, the resolution of the quantizer is defined by the number of bits it uses. An 8-bit quantizer will divide the input level into 256 steps. A 12-bit quantizer will divide the input level into 4,096 steps. Thus, a 12-bit quantizer has a higher resolution than an 8-bit quantizer. The number of steps for the quantizer is 2^n where n corresponds to the number of bits used. Quantizers (or ADCs) are commercially available from 4 to 24 bits. The required resolution is dictated by the application. There are literally hundreds of ADCs to choose from, and generally cost increases with resolution.

An important point to make is that the maximum value of the digital code of an ADC, namely all 1s (ones), does not correspond with the analog Full Scale (FS) but rather, one Least Significant Bit (LSB) less than full scale or:

\[ \text{Maximum value of digital code} = FS \times (1 - 2^{-n}) \]

For example, a 12-bit ADC with a 0 to ±10V analog range has a maximum digital code of 0x4000 (4095) and a maximum analog value of ±10V × (1 - 2^{-12}) or ±9.99756V. In other words, the maximum analog value of the converter never quite reaches the point defined as full scale. At any point of the input range of the ADC, there is a small range of analog values within which the same code is produced. This small range in values is known as the quantization size, or quantum. The quantum in Figure 10.2 is 1.25V and is found by dividing the full-scale analog range by the number of steps of the quantizer. Q is thus given by the following equation:

\[ Q = \frac{FSV}{2^n} \]

Q is the smallest analog difference that can be distinguished by the quantizer.
FSV is the full scale voltage range.
n corresponds to the number of bits used by the quantizer (i.e., ADC).
As shown in Figure 10.2 (Quantizer Error), a smooth error function is obtained if the ADC input is moved through its range of analog values and the difference between output and input is taken. For example, any voltage between 1.875V and 3.125V will produce the binary code 010.

All ADCs require a small but significant amount of time to quantize an analog signal. The time it takes to make the conversion depends on several factors: the converter resolution, the conversion technique, and the technology used to manufacture the ADC. The conversion speed (how fast an analog voltage is converted to digital) required for a particular application depends on how fast the signal to be converted is changing and on the desired accuracy. The conversion time (inverse of conversion speed) is frequently called aperture time. If the analog signal to measure varies twice by more than the resolution of the quantizer during the conversion time, then a sample-and-hold circuit should be used. ADCs are available with conversion speeds ranging from about three conversions per second to well over 100 million conversions per second.

10.01 Reading an ADC

The method used to read the ADC depends on how fast the ADC converts an analog voltage to a binary code. In most cases, however, the ADC must be explicitly triggered to perform a conversion. In other words, you must issue a command to the ADC to start the conversion process. Very fast ADCs, those that can convert an analog signal in less than 1 μs, generally have dedicated hardware to handle the fast conversion rate and will typically buffer the samples. When the buffer is full, the analog samples are processed offline. This is basically how a digital storage oscilloscope works. At the other end of the spectrum, ADCs used in voltmeters are generally slow (about 200 ms) but accurate (4 1/2 digits or 0.005 percent).

The actual method used to read an ADC depends on many factors: the conversion time of the ADC, how often you need the analog value converted, how many channels you have to read, etc. The next three sections describe some possible methods of reading an ADC.

70.01.01 Reading an ADC, Method #1

The scheme shown in Figure 10.3 assumes that the ADC conversion time is relatively slow (greater than about 5 ms). Here a driver (a function) has an analog input channel and returns the result of the conversion to your application. Your application tells the driver in Figure 10.3 and passes it the desired channel to read. The driver starts by selecting (through the multiplexer) the desired analog channel (Ω) to read. Before starting the conversion, you may want to wait a few microseconds to allow for the signal to propagate through the multiplexer and stabilize. If you don’t wait for the multiplexer’s output to stabilize, your readings may be unstable. Next, the ADC is triggered to start the conversion (∈). The driver then delays the conversion to complete (δ). Note that the delay time must be longer than the conversion time of the ADC. After the delay, the driver assumes that the conversion is complete and reads the ADC (δ). The binary result is then returned to your application (δ). The pseudocode is:
ReadAnalogInputChannel (Channel #)
{
    Select the desired analog input channel;
    Wait for MSS output to stabilize;
    Start ADC conversion;
    Delay 'x' ms to allow for conversion to complete;
    Read ADC and return result to the caller;
}

Figure 10.3  Reading an ADC (Method #1).

This method is simple and can be used with slow-changing analog signals. For example, you can use this method when measuring the temperature of a room (which doesn’t change very quickly).

10.01.02 Reading an ADC, Method #2

You can actually use a signal provided by most ADCs (i.e., the End Of Conversion (EOC) signal) to tell your driver when the ADC has completed its conversion. The code and your hardware in this case will be a little more complicated, but this method is more efficient.
Figure 10.4 Reading an ADC (Method #2).

Again, your application calls the driver by passing it the analog input channel to read. The driver shown in Figure 10.4 starts by selecting (through the multiplexer) the desired analog channel (3). At this point, you should again wait a few microseconds to allow for the signal to propagate through the multiplexer and stabilize. The ADC is then triggered to start the conversion (4). The driver then waits for a semaphore (5) with a timeout. A timeout is used to detect a hardware malfunction. In other words, you don’t want the driver to wait forever if the ADC fails (i.e., never finishes the conversion). When the analog conversion completes, the ADC generates an interrupt (6). The ADC conversion-complete ISR signals the semaphore (5), which notifies the driver that the ADC has completed its conversion. When the driver gets to execute, it reads the ADC (6) and returns the binary result to your application (7).

The pseudocode for both the driver and the ISR follows.

You would use this method if the conversion time of the ADC is greater than the execution time of the ISR and the call to wait for the semaphore. For example, your ADC takes 1 mS to perform a conversion, and the total execution time of the ISR and the call to wait for the semaphore requires only about 50 µs. If the execution time of the ISR and the call to wait for the semaphore is greater than the conversion time of the ADC, you might as well wait in a software loop (polling the ADC’s EOC line) until the ADC completes its conversion. This method will be discussed next.
Chapter 10: Analog I/Os — 333

ReadAnalogInputChannel (Channel #)
{
    select the desired analog input channel;
    wait for MX output to stabilize;
    start ADC conversion;
    wait for signal from ADC ISR (with timeout);
    if (timed out)
        signal error;
    else
        read ADC and return result to the caller;
    }

Conversion complete ISR
{
    signal conversion complete semaphore;
}

10.01.03 Reading an ADC, Method #3

The third method can be used if the conversion time of the ADC is less than the time needed to process the interrupt and wait for the semaphore, as described in the previous method. For example, depending on the microprocessor, an ADC with a conversion time less than 25 μs cannot afford the overhead of an interrupt and a semaphore which could take over 50 μs. In other words, the execution time to handle the interrupt overhead and the time to signal and wait for the semaphore can take more than 25 μs. This is true of most 8-bit and some 16-bit microprocessors.

You application calls the driver shown in Figure 10.5 by passing it the desired analog input channel to read. The driver starts by selecting (through the multiplexer) the channel to read (3). Again, before starting the conversion, you may want to wait a few microseconds to allow for the signal to propagate through the multiplexer and stabilize. The ADC is then triggered to start the conversion (3). The driver then waits (9) in a software loop for the ADC to complete its conversion. While waiting in the loop, the driver monitors the status (the EOC) or the BUSY signal of the ADC. You need to ensure that you have a way to prevent an infinite loop if your hardware becomes defective. An infinite loop is avoided by using a software counter which is decreased every time through the polling loop (see the pseudocode following this paragraph). The initial value of the counter is determined from the execution time of each iteration of the polling loop. For example, if you have an ADC that should perform a conversion in 50 μs and each iteration through the polling loop takes 5 μs, you will need to load the counter with a value of at least 10. You want to use the loop counter as an indication of a hardware malfunction and not to indicate when the ADC is done converting. Based on experience, you should load the loop counter so that a timeout occurs when the polling time exceeds the ADC conversion by about 25 to 50 percent. In other words, you would load the counter with a value between 13 and 15 in my example. When the ADC finally signals an end of conversion, the driver reads the ADC (8) and returns the binary result to your application (9).
The pseudocode for the driver is:

```c
ReadAnalogInputChannel (ChannelID)
{
    Select the desired analog input channel (i.e., MUX);
    Wait for MUX output to stabilise;
    Start ADC conversion;
    Load timeout counter;
    while (ADC Busy && Counter > 0) /* Polling Loop */
        if (Counter == 0) { /* Check for hardware malfunction */
            Signal error;
        } else {
            Read ADC and return result to the caller;
        }
}
```

Actually, I prefer this method because:

- You can get fairly inexpensive fast ADCs (~25 μs conversion time).
- You don’t have the added complexity of an ISR.
- Your signal has less time to change during a conversion.
- This method imposes very little overhead on your CPU.
- The polling loop can be interrupted to service interrupts.
10.01.04 Reading an ADC, Miscellaneous

The nice thing about reading analog input channels through drivers is that the implementation details are hidden from your application. You can use any of the three drivers shown without changing your application code.

By always returning the same number of bits to your application, you can make your application insensitive to the actual number of bits of the ADC. In other words, if the ADC driver always returned a signed 16-bit number irrespective of the actual number of bits for the ADC, your application would not have to be adjusted every time you changed the word size of your ADC. This is actually quite easy to accomplish, as shown in Figure 10.6. All you need to do is to shift left the binary value of the ADC until the most significant bit of the ADC value is in bit position number 14 of the result. I use a 16-bit signed result because the computations required to scale the result of the ADC need to be signed. This will be described in the next section. If you deal with higher resolution ADCs, you may want to write your drivers and application code to assume signed 32-bit values.

Figure 10.6 ADC driver always returning a signed 16-bit result.

For example, an 8-bit ADC can measure a voltage between 0 and 0.996094 (355/326) of the full scale voltage (see Equation [10.1]). This is the same as (255 << 7) / 32768, or 0.996894. Similarly, a 12-bit ADC can measure a voltage between 0 and 4095/4096 or 0.999756, which is the same as (4095 << 3) / 32768 (i.e., 0.999756). You can thus hide the details about how many bits each ADC has with respect to your application without losing any accuracy.
10.02 Temperature Measurement Example

As we have seen, an ADC produces a binary code based on a full scale voltage. If you are measuring a temperature, for example, this information means very little to you. What you really want to know is the temperature of what you are measuring. The circuit in Figure 10.7 shows a commonly used temperature sensor Integrated Circuit (IC), the National Semiconductor LM34A.

**Figure 10.7 Temperature measurement using an LM34A.**

The LM34A produces a voltage that is directly proportional to the temperature surrounding it, specifically, 10 mV/°F. Note that you can also obtain the temperature in degrees Celsius by using an LM35A. The amplifier is designed to have a gain of 2.5, and thus -50 to 300 °F will produce a voltage of -1.25 to 7.50 volts. By using a 10-bit ADC, you can obtain a resolution of about 0.342 °F (350 °F/1024). Note that the ADC can only convert positive voltages, and thus a bias of 1.25 volts is inserted following the amplification stage to ensure that a positive voltage is present at the input of the ADC for the complete temperature range. With this bias, -50 °F will appear as 0 V, 0 °F will be 1.25 V and 300 ºF will be 8.75 V. The value obtained at the ADC is given by:

\[
ADC_{counts} = \left( \frac{Temperature \, (^\circ F)}{V_{scale}} \right) \times 0.01 \times 2.5 \times V_{bias} + 1.25 \times V_{scale} + 1023
\]

counts is an industry standard convention that means the binary value of the ADC,

0.01 corresponds to the transducer transfer function — 10 mV/°F — specified by National Semiconductor.

2.5 is the gain of the amplifier stage and is established by the hardware designer.

1.25 is the bias voltage to ensure that the ADC always reads a positive voltage.

1023 is the maximum binary value taken by a 10-bit converter.

\( V_{scale} \) is the full scale voltage.

For example, a temperature of 100 °F would have a value of 383 counts (actually, 383.625). Note that the ADC can produce only integer values, and thus the actual value of 383.625 is truncated to 383.

To obtain the temperature read at the sensor, you need to rearrange Equation (10.3) so that temperature
is given as a function of ADC counts, as shown in Equation 10.4. This process is often called converting ADC counts to engineering units (E.U.):

\[
\text{Temperature} (^\circ F) = \frac{\text{ADC counts} \times 50^\circ F/\text{counts} - V_{\text{bias}}}{1023 \text{ counts} \times \frac{25.5}{V_{\text{ref}}}}
\]

The general form for this equation is:

\[
E.U. = \frac{\text{ADC counts} \times FSV}{(2^n - 1) \times V_{\text{bias}}} \times \frac{\text{Transducer}}{V/\text{EU}} \times A_y
\]

Where:
- **E.U.** is the engineering unit of the transducer (°F, PSI, Feet, etc.).
- **V_{\text{bias}}** is the bias voltage added to the output of the amplifier stage to allow the ADC to read negative values.
- **FSV** is the full-scale voltage of the ADC.
- **Transducer** corresponds to the number of volts produced by the transducer per engineering unit.
- **n** is the resolution of the ADC (in number of bits).

You can also write Equation 10.5 as follows:

\[
E.U. = \frac{\text{ADC counts} - \text{Bias counts}}{V/\text{EU}} \times \frac{FSV}{(2^n - 1) \times A_y}
\]

In this case, **Bias counts** corresponds to the ADC counts of the bias voltage as is given by the following equation:

\[
\text{Bias counts} = \frac{V_{\text{bias}}}{FSV} \times (2^n - 1)
\]

Note that most of the terms in Equation 10.6 are known when the system is designed, and thus, to save processing time, they should not be evaluated at run time. In other words, you could rewrite the equation as follows:

\[
E.U. = \frac{\text{ADC counts} - \text{ConvOffset counts}}{(\text{EU/counts}) \times \text{ConvGain}_{\text{(EU/counts)}}}
\]

where:

\[
\text{ConvGain}_{\text{(EU/counts)}} = \frac{FSV}{\text{Transducer} \times V/\text{EU}} \times A_y \times (2^n - 1)
\]

Note that the units of the conversion gain (ConvGain) are E.U. per ADC count.
\[ (10.10) \quad \text{ConvOffset}_{\text{counts}} = \left( \frac{V_{\text{bias}} \times (2^N - 1)}{FSV} \right) \]

In the temperature measurement example, the conversion gain would be 0.391007 and the conversion offset would be 127.875. You can apply fixed-point arithmetic and scale factors (see Chapter 5) to the temperature measurement example. The temperature of the LM34A is then given by:

\[ (10.11) \quad \text{Temperature (°F)} = \left( \frac{\text{ADC}_{\text{counts}} \times \text{ConvOffset}_{\text{counts}}}{\text{ConvGain}} \right) \times \text{TempScale} \]

Remember that you have a 10-bit ADC, and thus the range of counts is from 0 to 1023. You can scale this number by multiplying the ADC counts by 32 (shifting left five places). To perform the subtraction with the bias, you need to scale the bias by the same value, or 127.875 \times 32 = 40925-3. The gain (0.391007) can be scaled by multiplying by 55456, and thus the conversion gain is 256325-16. The temperature is then given by:

\[ (10.12) \quad \text{Temperature (°F)} = \left( \frac{\text{ADC}_{\text{counts}} \times 55456}{55456-5} \times 256325-16 \right) \]

or

\[ (10.13) \quad \text{Temperature (°F)} = \left( \frac{\text{ADC}_{\text{counts}} \times 55456}{55456-5} \times 256325-16 \right) \]

From Equation (10.3), 150°F would produce 511 ADC counts. Substituting 511 counts in Equation (10.12) produces the following:

\[ \text{Temperature (°F)} = 163525-5 \times 256325-16, \text{ or} \]

\[ \text{Temperature (°F)} = 3441663308-5 + 149.80 \]

or using Equation (10.13):

\[ \text{Temperature (°F)} = 958756-6 (\text{i.e., } 149.80) \]

The C code to convert the ADC counts to temperature is:

```c
#include <stdio.h>

int conv(char count) {
    int adc_count;
    int temp;

    adc_count = (int)(count << 5) - 4092;
    temp = (int)(163525 * (int)count * 256325) >> 32;
    return (temp); /* Result is scaled to 6 bits */
}
```

Note that `conv` corresponds to the ADC counts (10 bits). The total counts (counts) number is computed separately because a good compiler should perform this operation using 16-bit arithmetic instead of 32-bit (which would be faster). Counts and gain are then converted to INT32 because the multiplication needs 30-bit precision. The result is divided by 32768 so that it fits back into a 16-bit signed vari-
able. Finally, the temperature is returned in °F scaled S-6. You could obtain the temperature to the nearest degree by first adding 32 (0.5) and then dividing the result by 64. In other words, by rounding the result.

The electronic components used to provide the amplification and the bias voltage are generally inaccurate. On top of that, extra components can be added to allow the amplification stage and bias voltage to be precisely adjusted (this is, calibration). Adding such components, however, adds recurring cost to your system. Component inaccuracies easily can be compensated in software by modifying Equation [10.8] as:

\[ EU = \left( \frac{ADC_{counts} \times ConvOffst_{counts} \times Cal(Offset \times counts)}{EU \times (counts)} \right) \times ConvGain \]

The calibration gain (ConvGain) and calibration offset (Cal(Offset)) would be entered by a calibration technician using a keyboard/display or through a communications port. Both calibration parameters could then be stored in a non-volatile memory device such as battery backed-up RAM, EEPROM or even a floppy disk. The adjustment range of the calibration parameters is based on the accuracy of the electronic components used. A 10 percent adjustment range should be sufficient for most situations. For the calibration gain, we usually need an adjustment range between 0.90 (147485-14) and 1.10 (130725-14). In our example, all we need is an adjustment range between -100 (-329805-5) and +100 (329805-5) for the calibration offset when using a 10-bit ADC. The new C code to convert raw ADC counts to a temperature is:

```c
INT16S readTemp(int16S raw)
{
    INT16S cnts;
    INT16S temp;

    cnts = (raw << 5) - 4092 - Cal(Offset);
    temp = ((INT16S) ((INT16S) cnts * (INT16S) 25625) >> 15L);
    temp = ((INT16S) ((INT16S) temp * (INT16S) CalGain) >> 14L);
    return (temp); /* Result is scaled S-6 */
}
```

For example, if the actual gain of the amplification stage of our temperature measurement example was 2.45 instead of 2.50, then, CalGain would be set to 1.020408 (16785-14). Similarly, if the bias voltage was 1.27V instead of 1.25V then, you would have to subtract 0.02V, or 65 counts (see Equation [10.10]). In other words, Cal(Offset) would be set to -65S-5.
10.03 Analog Outputs

A typical digital to analog system generally consists of the following circuit elements:

- digital to analog converter (DAC)
- filter
- amplifier
- transducer

Digital-to-analog converters (DACs) are generally inexpensive devices, and thus each analog output channel can have its own DAC, as shown in Figure 10.8. The DAC converts a binary value provided by a microprocessor to either a current or a voltage (depending on the DAC). The voltage or current is filtered to smooth out the step change. An amplifier stage is sometimes used to increase the amplitude or power drive capability of the analog output channel in order to properly interface with the transducer. The transducer is used to convert the electrical signal to a physical quantity. For example, transducers are available to convert electrical signals to pressures (known as current-to-pressure transducers, or I to P). These pressures can be — and often are — used to control other physical devices.

**Figure 10.8 Digital-to-analog conversion.**

DACs are commercially available with resolutions from 4 to 16 bits. The resolution to choose from is application specific. There are literally hundreds of DACs to choose from. Generally, the cost of DACs increases with resolution and conversion speed. DACs are much faster than ADCs. Conversion time (also called settling time) is always less than a few microseconds and can be as fast as 3 nS (nano-
second). Very fast DACs are used in video applications, and because of their higher cost and lower resolution (8 bits), very fast DACs are seldom used in industrial applications.

A digital-to-analog conversion is handled exclusively in hardware. From a software standpoint, updating a DAC is as simple as writing the binary value to one or more (if more than 8 bits) I/O port locations or memory locations (when DACs are memory mapped).

**10.04 Temperature Display Example**

Suppose you wanted to display the temperature read by your LM35A (see Section 10.02) on a meter, as shown in Figure 10.9.

An 8-bit DAC is deemed sufficient considering the accuracy of these types of meters. The DAC is followed by a circuit that converts the voltage output of the DAC to a current (a V-to-I Converter). The Full Scale Voltage (FSV) of the DAC is set to 2.5 volts. The current converter is designed to produce about 42 μA/V, and the meter requires 100 μA for full scale. Your task is to write a function that takes the temperature (in °F) as an input and produces the proper output current (0 to 100 μA) to drive the meter.

**Figure 10.9 Temperature display.**

The relationship between the temperature and the meter current is shown in Figure 10.10.

**Figure 10.10 Temperature to DAC counts scaling.**

The graph can also be represented by the following linear equation:
\[ y = m \times x + b \]
where \( m \) is the slope and \( b \) is the \( Y \)-intercept (the value on the \( y \)-axis when \( x = 0 \)). The slope gives us the current per degree of temperature and is given by:

\[
\begin{align*}
\frac{m}{10} &= \frac{\left( Y_2 - Y_1 \right)}{\left( X_2 - X_1 \right)} \\
Y_0 &= m \times \left( X_0 - X_1 \right) + Y_1
\end{align*}
\]

In this case, the slope is 100\( \mu \)A/330°F, or 0.285714 \( \mu \)A/°F. The \( Y \)-intercept (i.e., \( Y_0 \)) is given by:

\[
Y_0 = m \times \left( X_0 - X_1 \right) + Y_1
\]

By substituting the values of \( m \), \( Y_1 \), \( X_1 \), and \( X_0 \) (i.e., 0) in Equations [10.17], we obtain a \( Y \)-intercept of 14.285714 \( \mu \)A. The meter current then is given by:

\[
\begin{align*}
\text{Meter}_{\mu A} &= 0.285714 \div \left( \frac{\mu A}{°F} \right) \times \text{Temperature}_{\circ F} + 14.285714 \, \mu A \\
\text{The meter current is also given by:}
\end{align*}
\]

\[
\begin{align*}
\text{Meter}_{\mu A} &= \frac{\text{DAC}_{\text{counts}} \times \text{FSV}}{256} \div \left( \frac{\mu A}{°F} \right) \\
\text{DAC}_{\text{counts}} &= \frac{256 \times 2.5}{42} \text{ Temperature}_{\circ F} + 14.285714 \, \mu A \\
\text{DAC}_{\text{counts}} &= \frac{\text{DAC}_{\text{counts}}}{256} \times \left( \frac{\mu A}{°F} \right) \text{ Temperature}_{\circ F} + 14.285714 \, \mu A
\end{align*}
\]

Combining Equations [10.18] and [10.19], I obtain:

\[
\begin{align*}
0.285714 \div \left( \frac{\mu A}{°F} \right) \times \text{Temperature}_{\circ F} + 14.285714 \, \mu A &= \frac{\text{DAC}_{\text{counts}} \times 2.5}{256} \div \left( \frac{\mu A}{°F} \right) \\
\text{DAC}_{\text{counts}} &= \frac{256 \times 2.5}{42} \text{ Temperature}_{\circ F} + 14.285714 \, \mu A
\end{align*}
\]

Solving for \( \text{DAC}_{\text{counts}} \), I obtain:

\[
\text{DAC}_{\text{counts}} = \int \frac{0.285714 \times 256}{2.5 \times 42} \div \left( \frac{\mu A}{°F} \right) \times \text{Temperature}_{\circ F} + 14.285714 \times 256
\]

Note that \( \int \) means that only the integer portion of the result is retained. As you can see, Equation [10.21] is also a linear equation, where \( m \) is 0.696596 and \( b \) is 34.829931. \( \text{DAC}_{\text{counts}} \), thus are given by:

\[
\text{DAC}_{\text{counts}} = \int \left( 0.696596 \div \left( \frac{\mu A}{°F} \right) \times \text{Temperature}_{\circ F} + 34.829931 \, \mu A \right)
\]

Substituting -20° F in Equation [10.22], I obtain \( 0 \) counts (as it should). Similarly, substituting 900° F in Equation [10.22], I obtain 243 counts, which should produce 100 \( \mu A \).

As with analog inputs, the electronic components used in circuits such as the voltage-to-current converter are generally inaccurate. You can compensate for component inaccuracies in software by modifying Equation [10.22] as:
Chapter 10: Analog I/Os — 343

\[ \text{DAC}_{\text{cont}} = \text{INT}(0.69659P_{\text{cont}}) \times \text{Temperature} + \text{Gain} \times 34.829931 + \text{Offset} \]

The effect of the calibration gain and offset is shown in Figure 10.11, which has been exaggerated for sake of discussion. The actual curve that you get from an incorrect gain and offset needs to be adjusted, as shown in Figure 10.11.

Figure 10.11 Calibration gain and offset adjustments (exaggerated).

The adjustment range of the calibration parameters is based on the accuracy of the electronic components. Based on experience, a 10 percent adjustment range should be sufficient for most situations. For the calibration gain, you only need an adjustment range between 0.90 and 1.10. For the calibration offset, you need an adjustment range between -25 and +25 for an 8-bit ADC. What would happen if the voltage-to-current converter was actually putting out 40 μA instead of 42 (a 5 percent error)? In this case, the slope in Equation (10.23) (see Equation (10.21), substituting 40 instead of 42) would need to be adjusted to 0.731428 and the intercept would need to be 36.571428. This can be accomplished by setting \( \text{Gain} \) and \( \text{Offset} \) to 1.05 and 1.741497, respectively.

The general form for Equation (10.23) is:

\[ \text{DAC}_{\text{cont}} = \text{INT}(\text{Gain}_{\text{cont}}) \times \text{Input}_{\text{cont}} + \text{Offset}_{\text{cont}} + \text{ConvGain} \times \text{Input}_{\text{cont}} + \text{ConvOffset}_{\text{cont}} \]
10.05 Analog I/O Module

In this chapter, I provide you with a complete analog I/O module that will allow you to read and scale up to 250 analog inputs and read and update up to 250 analog output channels. Each analog input channel is sampled at a regular interval and the scan rate for each channel can be programmed individually. This allows you to determine whether some analog inputs are scanned more often than others. Similarly, each analog output channel is updated at a regular interval and the update rate for each channel can also be programmed individually. This allows you to establish which analog outputs are to be updated more often.

The source code for the analog I/O module is found in the \SOFTWARE\BLOCKS\AOI\SOURCE\directory. The source code is found in the files AIO.C (Listing 10.1) and AIO.H (Listing 10.2). As a convention, all functions and variables related to the analog I/O module start with either AIO_ (functions and variables common to both analog inputs and outputs), A2_ (analog input functions and variables) or AO_ (analog output functions and variables). Similarly, #defines constants will either start with AIO_\_\_\_ or AO_.

10.06 Internals

The analog I/O module makes extensive use of floating-point arithmetic (additions, multiplications, and divisions). The reason I chose to use floating-point instead of integer arithmetic is that it is very difficult to make a general purpose analog I/O module using integer arithmetic. The analog I/O module can become CPU-intensive unless you have hardware-assisted floating-point (i.e., a math coprocessor). The analog I/O module, however, can be easily modified to make use of integer arithmetic if you have a dedicated application.

Figure 10.12 shows a block diagram of the analog I/O module. You should also refer to Listings 10.1 and 10.2 for the following description. As shown, the analog I/O module consists of a single task (AIOTask()) that executes at a regular interval (set by AIO_TASK_DELAY). AIOTask() can manage as many analog inputs and outputs as your application requires (up to 250 each). The analog I/O module must be initialized by calling AIOInit(). AIOInit() initializes all analog input channels, all analog output channels, the hardware (ADCs and DACs), a timer used to ensure exclusive access to the internal data structures used by the analog I/O module, and finally, AIOInit() creates AIOTask().

AIOI() is a data structure that contains configuration and run-time information for each analog input channel. An entry in AIOI[] is a structure defined in AIO.H and is called AIOI. AIOUpdate() is charged with reading all of the analog input channels on a regular basis. AIOUpdate() calls AIOI() and passes it a logical channel number (0...AOI_MAX_AI - 1). AIOI() is responsible for selecting the proper analog input through one or more multiplexers (based on the logical channel number), starting and waiting for the proper ADC to convert (if more than one is used), and for returning raw counts to AIOUpdate(). AIOI() is the only function that knows about your hardware, and thus AIOI() can easily be adapted to your environment.

NOTAI() is a data structure that contains configuration and run-time information for each analog output channel. Each entry in NOTAI[] also uses the AIO structure. AIOUpdate() is responsible for updating all of the analog output channels on a regular basis. NOTAIUpdate() calls NOTAI() and passes it a logical channel number (0...AOI_MAX_AO - 1) and the raw DAC counts. NOTAI() is responsible for outputting the raw counts to the proper DAC based on the logical channel. NOTAI() is the only function that knows about your hardware, and thus NOTAI() can easily be adapted to your environment.
Figure 10.12 AIO module flow diagram.

Figure 10.13 shows a flow diagram of a single analog input channel. Note that I used electrical symbols to represent functions performed in software. ADC ??? are all members of the AIO structure. AIOUpdate() updates each channel as described in the following paragraphs.

Figure 10.13 Analog input channel flow diagram.

The raw counts obtained from AI[0..3] are placed in the channel's .ADCHi variable. The raw counts are then added to .ADCOffset and .ADCOvGain. The result of this operation is then multiplied by .ADCalGain and .ADCOvGain. These mathematical operations are basically used to implement Equation 10.14: 
346 — Embedded Systems Building Blocks, Second Edition

\[ 10.25 \]

\[ \text{AIOScaleIN} = (\text{AIORev} + \text{AIOConvOffset} + \text{AIOcalOffset}) \times \text{AIOConvGain} \times \text{AIOcalGain} \]

\text{AIOcalOffset} is a pointer to a function that is executed when the channel is updated. The function allows you to apply further processing when reading an analog input. For example, a Resistance Temperature Detector (RTD) is a device that requires special processing. The temperature at the RTD is proportional to the resistance of the RTD (but is nonlinear). A scaling function can thus be written to convert \text{AIOcalGain} (the resistance of the RTD) to a temperature in degrees Fahrenheit (placed in \text{AIOcalOffset}). There are many types of RTDs, and thus you need to be able to specify the actual type used. This is where \text{AIOcalFactoryArg} comes in. \text{AIOcalFactoryArg} is a pointer to any arguments that your scaling function requires. In the case of an RTD, this argument can specify the type of RTD used. The scaling function that you write must be declared as:

\text{void AIOcal??? (AIO *p aio);}

When called, your scaling function will receive a pointer to the AIO channel to scale (or linearize). The input to your function is available in p aio->AIOcalGain, and your function must place the result in p aio->AIOcalOut. Any arguments to the scaling function are found through p aio->AIOcalFactoryArg. If you do not have any linearization function, the value of \text{AIOcalOut} is simply copied to \text{AIOcalOut} by AIOcalArg.

\text{AIOcalArg} is a software switch that is used to prevent the analog input from being updated. This feature allows your application code to "bypass" the channel and force a value into \text{AIOcal}. When another part of your application code tries to read the analog input channel, it will actually be getting the forced value instead of what the sensor is measuring. I have found this feature to be invaluable.

\text{AIOcalArg} is the value that your application code will obtain when it needs the latest value read by the analog input channel (by calling AIOGet()). AIOcalArg containing engineering units. This means that if the analog input channel monitors a pressure, your application code will obtain a value in either psi, kPa, bar, inHg, etc.

\text{AIOpassCtrl} allows your application code to specify how often the analog input channel is to be updated. In fact, \text{AIOpassCtrl} specifies how many analog input scans are needed before the channel is updated. In other words, if analog inputs are read every 50 ms and you specify a pass count of 20, then the analog input channel will be read every 1000 ms (i.e., 1 second).

Figure 10.14 shows a flow diagram of a single analog output channel. Note that I used electrical symbols to represent functions performed in software. As with analog input channels, \text{AIO???} are all members of the AIO structure. AIOUpdate() updates each channel as described in the following paragraphs.
Figure 10.14 Analog output channel flow diagram.

Your application deposits the value for the analog output channel by calling AGSet(). This value is passed in engineering units. This means that if the analog output channel controls a meter that displays the RPM of a rotating device you call AGSet() by specifying an RPM and the analog output channels takes care of figuring out how much voltage or current is needed to display the RPM.

.A0BypassEn is a software switch used to override the value that your application code is trying to put out on the analog output channel. Another function provided by the analog I/O module is used to load .A0ScaleIn. This feature is very useful for debugging purposes because it allows you to test your output independently of the application code.

.A0ScalePct is a pointer to a function that is executed when the analog output channel is updated. The function allows you to apply further processing prior to updating an analog output. For example, a 0 to 100 mA output may be controlling a valve. If the flow through the valve is proportional to the output — but nonlinear, the function can make the valve action look linear with respect to your application. If your software needs to support different types of valves, you can specify which valve is being used through .A0ScalePctArg. .A0ScalePctArg is a pointer to any arguments that your scaling function requires. The scaling function that you write must be declared as follows:

```c
void A0ScalePct???(A0 *pA0);
```

When called, your scaling function will receive a pointer to the A0 channel to scale (or linearize). The input to your function is available in pA0->A0ScaleIn, and your function must place the result in pA0->A0ScaleOut. Any arguments to your function are found...
through paio->AIOScaleFunctArg. If you do not have any linearization function, the value of
AIOScaleIn is simply copied to AIOScaleOut by AIOUpdate().
AIOScaleOut is then multiplied by AIOCalcGain and AIOConvGain. The result of the multi-
plification is added to AIOConvOffset and AIOCalcOffset. The result of this operation is depos-
ted in AIORaw so that it can be sent to the proper DAC by AOOut().

\[ AIORaw = \frac{AIOScaleOut \times AIOConvGain \times AIOCalcGain}{AIOConvOffset + AIOCalcOffset} \]

AIOLim is used to ensure that AIORaw does not exceed the maximum counts allowed by the
DAC. For example, an 8-bit DAC has a range of 0 to 255 counts. An output of 256 counts to a DAC
would appear to the DAC as 0 (the lower eight bits of 10000000). AIOLim contains the maximum
count that can be sent to the DAC (255 for an 8-bit DAC).
AIOPassCnts allows your application code to specify how often the analog output channel is to
be updated. In fact, AIOPassCnts specifies how many analog output scans are needed before the
channel is updated. In other words, if analog outputs are updated every 50 ms and you specify a pass
count of 5, the analog output channel will only be updated every 250 ms.

### 10.67 Interface Functions

Your application software knows about the analog I/O module through the interface functions shown in
Figure 10.15.

**Figure 10.15 Analog I/O module interface functions.**
AICfgCal

AICfgCal() is used to set the calibration gain and offset of an analog input channel. The analog I/O module implements Equation [10.14], and this function is used to set the value of CalGain and CalOffset.

Arguments

m is the desired analog input channel to configure. Analog input channels are numbered from 0 to AIO_MAX_AI - 1.

gain is a multiplying factor that is used to compensate for component inaccuracies and does not have any units. The gain would be entered by a calibration technician and stored in some form of non-volatile memory device such as an EEPROM or battery-backed up RAM.

offset is a value that is added to the raw counts of the ADC to compensate for offset type errors caused by component inaccuracies. The offset would also be entered by a calibration technician and stored in some form of non-volatile memory device such as an EEPROM or battery-backed up RAM.

Return Value

AICfgCal() returns 0 upon success and 1 if the analog input channel you specified is not within 0 and AIO_MAX_AI - 1.

Notes/Warnings

None

Example

```c
void main(void)
{

  /* Calibration gain and offset obtained by technician */
  AICfgCal(16, 0x1011.095, 0x2203.01)
}
```
AIConfigConv() is used to set the conversion gain, offset, and the value of the pass counter for an analog input channel. The analog I/O module implements Equations [10.14] and this function is used to set the value of ConvGain and ConvOffset.

**Arguments**

- \( n \) is the desired analog input channel to configure. Analog input channels are numbered from 0 to \( \text{AI1,MAXAI} \) = 1.
- \( \text{gain} \) is the conversion gain of the ADC channel in engineering units per count \( \text{EU/} \text{count} \). \( \text{gain} \) is given by Equation [10.9] which is repeated in Equation [10.27] for your convenience:

\[
\text{gain}_{(\text{EU}/(\text{count})} = \frac{\text{FSV}}{\frac{\text{Transducer}}{\text{V(EU)}} \times A_y \times (2^{\text{bits}} - 1)}
\]

- \( \text{FSV} \) is the Full Scale Voltage of the ADC and typically is the reference voltage used with the ADC.
- \( \text{Transducer} \) corresponds to the number of volts produced by the transducer per engineering unit. For example, the LMI40 produces 0.01 volt per degree Fahrenheit.
- \( A_y \) is the gain of the amplifier stage of an analog input channel (see Figure 10.1).
- \( \text{bits} \) is the number of bits of the ADC.
- \( \text{offset} \) is used to bias the ADC counts. \( \text{offset} \) is given by Equation [10.10] which is repeated in Equation [10.28] for your convenience.

\[
\text{offset}_{(\text{counts})} = \frac{V_{\text{bias}} \times (2^{\text{bits}} - 1)}{\text{FSV}}
\]

- \( V_{\text{bias}} \) is the bias voltage added to the output of the amplifier stage to allow the ADC to read negative values (see Figure 10.7 on page 336 for an example or how to use the \( \text{bias} \)).
- \( \text{pass} \) is used to specify a pass count. The pass count specifies the module how often the analog channel will be read. The analog I/O module reads all analog input channels on a regular basis every so many clock ticks. This is called scanning. \( \text{pass} \) specifies how many scans are needed to read the analog input channel. For example, suppose the analog I/O module's scan rate is 10 Hz and you specify a pass count of 5 for analog input channel 0, Analog input channel 0 will be read every half second. I included a pass count because some analog input channels may not need to be read as often as others. For example, if you wanted the program to read the temperature of a room, you could tell it to read the temperature every 250 scans (or every 25 seconds, as in my example).

**Return Value**

AIConfigConv() returns 0 upon success and 1 if the analog input channel you specified is not within 0 and \( \text{AI1,MAXAI} \) = 1.
Notes/Warnings
None

Example

```c
void main (void)
{
    // Conversion gain and offset obtained by hardware engineer
    AICfgConv(0, (FVoL)1.987, (FP32)123.0, 1);
}
```
AICfgScaling() is used to specify a scaling function to be executed when the analog input channel is
read. The scaling function allows you to apply further processing when reading an analog input. There
is no need to call AICfgScaling() if the analog input channel does not need a scaling function. In
fact, if you don’t define a scaling function the member .AIOScalingIn will simply be copied to
.AIOScalingOut by AIUpdate() (see code).

Arguments

n is the desired analog input channel to configure. Analog input channels are numbered from 0 to
AIO_MAX_AI - 1.

fnet is a pointer to the scaling function that will be executed when the analog input channel is read.
You must write fnet to expect an argument. Specifically, fnet must be written to receive a pointer to
the analog I/O data structure called AIO as shown in the code fragment following this paragraph. You
specify a NULL pointer to prevent a previously configured channel from using a scaling function:

    void fnet (AIO *p aio);

arp is a pointer to any arguments or parameters needed for the scaling function. This argument can be
used to specify specific options about the scaling being performed.

Return Value

AICfgScaling() returns 0 upon success and 1 if the analog input channel you specified is not within
0 and AIO_MAX_AI - 1.

Notes/Warnings

The scaling function is assumed to take its input from p aio->AIOScaleIn and produce its result in
p aio->AIOScaleOut.
Example

INT8U ThermoType = THermo_TVIK_c;

void main (void){
   
   AICfgScaling(0, ThermoLin, (void *)ThermoType);
   
}

void ThermoLin (AIO *palo)
{
   /* Function to linearize a thermocouple */
   palo->AIOScaleIn is assumed to contain the number of millivolts for
   the thermocouple.
   palo->AIOScaleOut is where the temperature of the thermocouple
   is assumed to be saved to.
   palo->AIOScaleFConfArg could have also indicated the type of
   thermocouple used as well as whether the temperature is in
   degrees F or C.
}
The current value of the analog input channel can be obtained by calling AIGet(). The value obtained is in engineering units or, physical units. For example, if the analog input channel is measuring a temperature from a thermocouple then the value returned is the number of degrees at the thermocouple.

**Arguments**

`n` is the desired analog input channel. Analog input channels are numbered from 0 to AIO_MAX_AI - 1.

`pval` is a pointer to where the value of the analog input channel will be stored.

**Return Value**

AIGet() returns 0 upon success and 1 if the analog input channel you specified is not within 0 and AIO_MAX_AI - 1.

**Notes/Warnings**

The value returned is the last 'scanned' value. In other words, an ADC conversion is not performed when you call this function — ADCScan() is responsible for 'scanning' the analog input on a continuous basis.

**Example**

```c
void Task (void *pdata)
{
  INT8U err;
  PP32 u;

  for (i); 
  
    err = AIGet(0, &u); /* Get current value of analog input 0 */
    
}
```
**AIOInit()**

```c
void AIOInit(void);
```

`AIOInit()` is the initialization code for the analog I/O module. `AIOInit()` must be called before you use any of the other analog I/O module functions. `AIOInit()` is responsible for initializing the internal variables used by the module and for creating the task that will update the analog inputs and outputs.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

You are expected to provide the value of the following compile-time configuration constants (see Section 10.08, "Analog I/O Module, Configuration"):

- `AIO_TASKStackSize`
- `AIO_TASKPRIO`
- `AIO_MAX_AI`
- `AIO_MAX_AO`

**Example**

```c
void main (void)
{
    ...
    AIOInit();
    ...
}
```
Your application software can bypass or override the analog input channel value by using this function. A1SetBypass() doesn’t do anything unless you open the bypass switch by calling A1SetBypassOn().

Arguments
n is the desired analog input channel to override. Analog input channels are numbered from 0 to A10_MAX_AI – 1.
val is the value you want A1Get() to return to your application. The value you pass to A1SetBypass() is in engineering units.

Return Value
A1SetBypass() returns 0 upon success and 1 if the analog input channel you specified is not within 0 and A10_MAX_AI – 1.

Notes/Warnings
A1SetBypass() forces the value of .AI0SU in Figure 10.13 when .A10Bypass201 is set to TRUE.

Example

```c
void task (void *pdata) {

    PP32 val;

    for (;;) {
        
        val = Get value from keyboard;
        A1SetBypass(0, (PP32)val);

    }
}
```
AISetBypassEn()

```
AISetBypassEn(unsigned int n, BOOLEAN state);
```

`AISetBypassEn()` allows your application code to prevent the analog input channel from being updated. This permits another part of your application to set the value returned by `AISet()`. In other words, you can "fool" your application code that monitors the analog input channel into thinking that the value is coming from a sensor, when in fact, the value returned by the analog input channel can come from another source. The value of the analog input channel is set by `AISetBypass()`. `AISetBypassEn()` and `AISetBypass()` are very useful functions for debugging.

**Arguments**

- `n` is the desired analog input channel to bypass. Analog input channels are numbered from 0 to `AIO_MAX_AT - 1`.
- `state` is the state of the bypass switch. When `TRUE`, the bypass switch is open (i.e., the analog input channel is bypassed). When `FALSE`, the bypass switch is closed (i.e., the analog input channel is not bypassed).

**Return Value**

`AISetBypassEn()` returns `0` upon success and `1` if the analog input channel you specified is not within the range `0` and `AIO_MAX_AT - 1`.

**Notes/Warnings**

`AISetBypassEn()` forces the value of `AIOBypassEn` in Figure 10.13.

**Example**

```c
void main(void)
{
    AISetBypassEn(0, 1);
}
```
AOCfgCal() is used to set the calibration gain and offset of an analog output channel. An analog output channel basically implements a generalization of Equation [10.23], as shown in Equation [10.29]:

\[
\text{DACout} = \text{INT} \left( \text{AIOConvGain} \times \text{AIOCalGain} \times \text{AIOScaleOut} + \text{AIOConvOffset} \times \text{AIOCalOffset} \right)
\]

You can specify a calibration gain (AIOCalGain) and offset (AIOCalOffset) to compensate for component inaccuracies.

**Arguments**

- **n** is the desired analog output channel. Analog output channels are numbered from 0 to \(\text{AIO_MAX_AO} - 1\).
- **gain** is a multiplying factor that is used to compensate for component inaccuracies and doesn't have any units. gain sets the value of AIOCalGain in Figure 10.13. The gain would be entered by a calibration technician and stored in some form of non-volatile memory device such as an EEPROM or battery-backed-up RAM.
- **offset** is a value that is added to the raw counts before outputting to DAC to compensate for offset-type errors caused by component inaccuracies. offset sets the value of AIOCalOffset in Figure 10.13. The offset would also be entered by a calibration technician and stored in some form of non-volatile memory device such as an EEPROM or battery-backed-up RAM.

**Return Value**

AOCfgCal() returns 0 upon success and 1 if the analog output channel you specified is not within 0 and \(\text{AIO_MAX_AO} - 1\).

**Notes/Warnings**

None

**Example**

```c
void main (void)
{

    AOCfgCal(0, (PF32)1.05, (PF32)10.6);

}
```
Chapter 10: Analog I/Os – 359

AOCfgConv()

AOCfgConv() is used to set the conversion gain, conversion offset, and the value of the pass counter for an analog output channel. An analog output channel basically implements a generalization of Equation [10.20], as shown in Equation [10.29] (see page 358). AOCfgConv() is used to set the value of .AOCfgConvGain and .AOCfgConvOffset.

Arguments

n is the desired analog output channel to configure. Analog output channels are numbered from 0 to ATO_MAX_AO - 1.

gain is the conversion gain for the analog output channel in counts per engineering unit (Counts/ELU). gain sets the .AOCfgGain field of Figure 10.14.

offset is used to bias the DAC counts and sets the .AOCfgOffset field of Figure 10.14.

limit is used to specify the maximum count that can be sent to the DAC. This argument ensures that the DAC will never be written with a count larger than limit. For example, an 8-bit DAC has a maximum count of 255 (2^8 - 1). limit sets the .AOCfgLim field of Figure 10.14.

pass is used to specify a pass count. The pass count is used to specify to the module how often the analog channel will be updated. The analog I/O module updates all analog output channels on a regular basis every so many clock ticks. This is called scanning. pass specifies how many scans we need to update the specific analog output channel. For example, suppose the analog I/O module scans rate at 10 Hz and you specify a pass count of 2 for analog output channel #4. In this case, analog output channel #4 will be updated five times per second. I included a pass count because some analog output channels may not need to be updated as often as others. pass sets the .AOCfgPass field of Figure 10.14.

Return Value

AOCfgConv() returns 0 upon success and 1 if the analog output channel you specified is not within 0 and ATO_MAX_AO - 1.

Notes/Warnings

None

Example

void main (void)
{
    .AOCfgConv(0, (FF32)(1.0), (FF32)(10.6), D0FF, 1);
}

AOCfgScaling() is used to specify a scaling function to be executed when the analog output channel is updated. The scaling function allows you to apply further processing before updating an analog output. You don't need to call this function if your analog output channel doesn't need a scaling function. In this case, the .AIOScaleIn field will simply be copied to the .AIOScalingOut field by AOCfgPlate() (see code).

Arguments

n is the desired analog output channel. Analog output channels are numbered from 0 to AIO_MAX_AO - 1.

fnct is a pointer to the scaling function that will be executed when the analog output channel is updated. fnct sets the value of .AIOScaleFnc in Figure 10.14. fnct must be written to receive a pointer to the analog I/O data structure called AIO as follows:

```c
void fnct (AIO *pAIO);
```

arg is a pointer to any arguments or parameters needed for the scaling function. arg sets the value of .AIOScaleFncArg in Figure 10.14. This argument can be used to specify specific options about the scaling being performed.

Return Value

AOCfgScaling() returns 0 upon success and 1 if the analog output channel you specified is not within 0 and AIO_MAX_AO - 1.

Notes/Warnings

The scaling function is assumed to take its input from pAIO->AIOScaleIn and produce its result in pAIO->AIOScaleOut.
Example

```c
void main (void)
{

  NOCTFgScaling(0, ActLin, (void *)&0);

}

void ActLin (AIO *pio)
{

  /* Linearize actuator function */
  pio->AIOScaleIn is the input value to the scaling function.
  pio->AIOScaleOut is where the scaling function will place the result.
  pio->AIOScalePortArg in this case is not used but could be made
  to tell ActLin() the type of actuator to linearize.

```
This function is used by your application software to set the value of the analog output channel. The value you set the channel to is specified in engineering units. In other words, if your analog output channel has been configured to control the position of a valve in percent, then you would pass the desired percentage of position you desire (a number between 0.0 and 100.0).

Arguments
- \texttt{n} is the desired analog output channel. Analog output channels are numbered from 0 to \texttt{AIO\_MAX\_AO - 1}.
- \texttt{val} is the desired value for the analog output channel and is specified in engineering units.

Return Value
- \texttt{AOSet()} returns 0 upon success and 1 if the analog output channel you specified is not within 0 and \texttt{AIO\_MAX\_AO - 1}.

Notes/Warnings
None

Example
```c
void Task (void *pdata)
{
    FF32 valve;

    for (;;) {
        
        // Get desired value position from user:
        AOSet(0, FF32(valve));
    }
}
```
**AOSetBypass()**

```c
IMVS AOSetBypass(DIPSW n, FP32 val);
```

Your application software can bypass or override the analog output channel value by using this function. `AOSetBypass()` doesn’t do anything unless you open the bypass switch by calling `AOSetBypassEn(1)`, as described previously. As with `AOSet()`, the value you set the channel to is specified in engineering units.

**Arguments**

- `n` is the desired analog output channel. Analog output channels are numbered from 0 to `AIO_MAX_AO - 1`.
- `val` is the value that you want to force into the analog output channel (in engineering units).

**Return Value**

`AOSetBypass(1)` returns 0 upon success and 1 if the analog output channel you specified is not within 0 and `AIO_MAX_AO - 1`.

**Notes/Warnings**

None

**Example**

```c
void Task (void *pdata)
{
    FP32 val;

    for (;;) {
        val = Get value from keyboard;
        AOSetBypass(0, (FP32)val);
    }
}
```
AOSetBypassEn() allows you to prevent your application from changing the value of an analog output channel. This allows you to gain control of the analog output channel from elsewhere in your application code. This is a quite useful feature because it allows you to test your analog output channels one by one. In other words, you can set an analog output to any desired value even though your application software is trying to control the output. The value of the analog output channel is set by AOSetBypass(). AOSetBypassEn() and AOSetBypass() are very useful for debugging.

Arguments
n is the desired analog output channel. Analog output channels are numbered from 0 to AIO_MAX_AO – 1.

state is the state of the bypass switch. When TRUE, the bypass switch is opened (i.e., the analog output channel is bypassed). When FALSE, the bypass switch is closed (i.e., the analog output channel is not bypassed).

Return Value
AOSetBypassEn()) returns 0 upon success and 1 if the analog output channel you specified is not within 0 and AIO_MAX_AO – 1.

Notes/Warnings
None

Example

```c
void main (void)
{
    
    AOSetBypassEn(0, TRUE);
    
}
```
10.08 Analog I/O Module, Configuration

Configuration of the analog I/O module is quite simple. 

1. You need to define the value of five #defines. The #defines are found in AIO.h (or CFG.H). 
   AIO_TASK_PRIO is used to set the priority of the analog I/O module task. 
   AIO_TASK_DLY is used to establish how often the analog I/O module will be executed. 
   AIO_TASK_DLY determines the number of milliseconds to delay between execution of the ana-
   log I/O task. 

   **WARNING**
   In the previous edition of this book, you needed to specify AIO_TASK_DLY_TICKS which speci-
   fied the number of ticks between execution of AIODTask(). Because µCOS-II provides a more 
   convenient function (i.e., osDelaySysTick()) to specify the task execution period in hours, min-
   utes, seconds and milliseconds, AIO_TASK_DLY_TICKS is no longer used and AIO_TASK_DLY 
   now specifies the scan period in milliseconds instead of ticks. 

   AIO_TASK_STK_SIZE specifies the size of the stack (in bus width units) allocated to the analog 
   I/O task. The number of bytes allocated for the stack is thus given by: AIO_TASK_STK_SIZE times 
   sizeof (OS_TASK). 

   **WARNING**
   In the previous edition of this book, AIO_TASK_STK_SIZE specified the size of the stack for 
   AIODTask() in number of bytes. µCOS-II assumes the stack is specified in stack width elements. 

   AIO_MAX_AI determines the number of analog input channels that will be handled by the analog 
   I/O task. 
   AIO_MAX_AO determines the number of analog output channels handled by the analog I/O task. 

2. You will need to define how analog inputs are read (i.e., how to read your ADC(s)). ADCs must all be 
   handled through AIRD(). The function prototype for AIRD() is: 

   int32_t AIRD(int32_t ch); 

   AIRD() is called by AIDUpdate() (see code) and is passed the logical channel number (0 to AIO_MAX_AI 
   – 1). You must translate this logical channel into code that selects the proper multiplexer for the desired 
   channel, start the ADC, wait for the conversion to complete, read the ADC, and finally, return the ADC’s 
   counts.
3. You will need to provide the code for the function that writes to all DACs (i.e., `AOInit()`). The function prototype for `AOInit()` is:

```c
void AOInit (void);
```

`AOInit()` is called by `ADUpdate()` (see code) and is passed the logical channel number (0 to `ALO_MAX_AO` - 1). You must translate this logical channel into code that selects the proper DAC for the desired channel. `AOInit()` is also passed the counts to send to the DAC. Your code must thus write the counts to the proper DAC.

4. You will need to provide the hardware initialization function (`AOInitIO()`) which is called by `AOInit()`. The function prototype for `AOInitIO()` is:

```c
void AOInitIO (void);
```

### 10.09 How to Use the Analog I/O Module

Let’s assume that you need to read the analog inputs and control the analog outputs shown in Figure 10.16.

**Figure 10.16 Using the analog I/O module.**

<table>
<thead>
<tr>
<th>Analog Inputs</th>
<th>Your Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM-34A (1 sec.)</td>
<td>Temperature (-50 to 200 °F, 1 °F)</td>
</tr>
<tr>
<td>100 ohm RTD (100 mΩ)</td>
<td>Temperature (-50 to 200 °F, 0.02%)</td>
</tr>
<tr>
<td>J-type Thermocouple (500 mΩ)</td>
<td>Temperature (-50 to 750 °F, 1 °F)</td>
</tr>
<tr>
<td>J-type Thermocouple (500 mΩ)</td>
<td>Temperature (-50 to 1000 °F, 1 °F)</td>
</tr>
<tr>
<td>Voltage (1 sec.)</td>
<td>Voltage (0 to 15V, 0.1V)</td>
</tr>
<tr>
<td>Pressure (100 mΩ)</td>
<td>Pressure (0 to 30 PSI, 0.1 PSI)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Your Application</th>
<th>Analog Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature (-50 to 200 °F, 1 °F, 100 mΩ)</td>
<td>Temperature sensor (0 to 100 µA)</td>
</tr>
<tr>
<td>(0 to 100%, 0.1%, 100 mΩ)</td>
<td>Fuel Control (4 to 20 mA)</td>
</tr>
<tr>
<td>(0 to 6000 RPM, 1%, 200 mΩ)</td>
<td>RPM meter (0 to 10 µA)</td>
</tr>
</tbody>
</table>

The analog I/O module has six analog inputs, and thus you will configure `ALO_MAX_AI` to 6. Similarly, to update three analog outputs, you need to set `ALO_MAX_AO` to 3. We can set `ATG_TASK_DLY` to 100 (i.e., milliseconds) because all analog I/Os need to be read or updated in multiples of 100 mS.
Chapter 10: Analog I/Os — 367

Obviously, you need to allocate sufficient stack space (i.e., AIO_TASK_STACK_SIZE) for AIOTask() as well as determine what priority (i.e., AIO_TASK_PRIO) you want to give to that task.

To initialize the analog I/O module, you need to call AIOInit() prior to using any of the analog I/O module functions. You would typically do this in main():

void main (void)
{
    OSInit();        /* Initialize the O.S. (uC/OS-II) */
    ...
    AIOInit();       /* Initialize the analog I/O module */
    ...
    OSStart();       /* Start multitasking (uC/OS-II) */
}

You would initialize each one of the analog I/O channels from an application task, as shown in the
code fragment following this paragraph. It is important that you do this at the task level because some of
the analog I/O module services assume that the operating system is running in order to access
the mutual exclusion semaphore (AIONe).  

void AppTask (void *data)
{
    data = data;
    /* Initialize analog I/O channels here ... */
    ...
    for (i; i;)
        /* Application task code ... */
}

Let’s assume the hardware designer came up with the circuit shown in Figure 10.17 to read the ana-
log inputs. As you can see, each input has signal conditioning circuitry which feeds into a multiplexer.
The multiplexer selects one of the analog inputs to be converted by a 12-bit analog-to-digital converter
(ADC).
Figure 10.17 Analog inputs.
10.09.01 How to Use the Analog I/O Module, AI #0

Analog input channel #0 is an LM-34A temperature sensor used to read temperatures from -50 to 200°F. Using Equation [10.9], the conversion gain is:

\[ \text{ConvGain} \left( \text{EU}/\text{count} \right) = \frac{FSV}{\text{Transducer} \times V_{(EU)} \times A_V \times (2^n - 1)} \]

\[ \text{ConvGain} \left( \text{FV}/\text{count} \right) = \frac{10}{0.01 \times V_{(FV)} \times 4 \times (2^{12} - 1)} \]

\[ \text{ConvGain} \left( \text{FV}/\text{count} \right) = 0.001450 \]

From Equation [10.10], the conversion offset is:

\[ \text{ConvOffset}_{\text{counts}} = \left( \frac{V_{\text{offset}} \times (2^n - 1)}{FSV} \right) \]

\[ \text{ConvOffset}_{\text{counts}} = \left( \frac{0.75 \times (2^{12} - 1)}{10} \right) \]

\[ \text{ConvOffset}_{\text{counts}} = -307.125 \]

The temperature at the LM34A is given by Equation [10.11] and is:

\[ \text{Temperature}_{F} = \left( \frac{\text{ADC}_{\text{counts}} + \text{ConvOffset}_{\text{counts}}}{\text{ConvGain}_{\text{EU}/\text{count}}} \right) \times 0.06^\circ \]

Because the LM34A only needs to be read once per second, the pass counter for the channel will be set to 10 (i.e., 10 × 100 mV scan period).

10.09.02 How to Use the Analog I/O Module, AI #1

Analog input channel #1 is a 100-ohm Resistance Temperature Device (RTD). The RTD has about 80 ohms of resistance when the temperature at the RTD is -50°F and 139 ohms when the temperature at the RTD is 200°F. Unfortunately, the temperature at the RTD is not a linear function of resistance, and thus you will have to write a linearization function (beyond the scope of this chapter). The current source is used to develop a voltage across the RTD so that the resistance of the RTD can be measured. The circuit produces 1 mV per ohm (which is before the amplifier). By using Equations [10.9],[10.10], and [10.11], the resistance of the RTD is given by:

\[ \text{ConvGain}_{\text{(ohms)/count}} = 0.034886 \]

\[ \text{ConvOffset}_{\text{(ohms)}} = -2293.2 \]
Resistance_{\text{ohms}} = \left( \frac{\text{ADC}_{\text{counts}}}{2293.2} \right) \times 0.034886

The pass counter for analog input channel #1 will be set to 1 in order to read the RTD every 100 mS.

10.09.03 How to Use the Analog I/O Module, AX #2

Analog input channel #2 is a J-Type thermocouple (another temperature measurement device). If you want to get the official reference on thermocouples, you should get the NIST Monograph 175 (see "Bibliography" on page 394). A thermocouple produces a small voltage (called the Seebeck voltage) that varies as a function of temperature. The temperature at the thermocouple is not a linear function of the voltage produced. To further complicate things, the temperature at the thermocouple is also a function of a reference temperature called the Cold junction. Determining the temperature at the thermocouple is beyond the scope of this book. Let's say for now that all you need to do is to measure the voltage (actually milli-volts) produced by the thermocouple. It is thus up to you to write a linearization function (also called thermocouple compensation function). A J-Type thermocouple produces -2.223 mV at 50 °F and 21.785 mV at 750 °F. This voltage is amplified by 400 so that it can be read by the ADC. A bias voltage is introduced to ensure that the ADC only sees positive voltages. From Equations [10.9], [10.10], and [10.11], the number of milli-volts at the thermocouple is given by:

\[
\text{ConvGain}_{\text{mV/(count)}} = 0.006105
\]

\[
\text{ConvOffset}_{\text{counts}} = -409.5
\]

\[
\text{Thermocouple}_\text{mV} = \left( \frac{\text{ADC}_{\text{counts}}}{-409.5} \right) \times 0.006105
\]

All you have to do is linearize the thermocouple based on the number of milli-volts read from the thermocouple. The pass counter for analog input channel #2 will be set to 5 in order to read the thermocouple every 500 mS.

10.09.04 How to Use the Analog I/O Module, AX #3

Analog input channel #3 is also a J-Type thermocouple. A J-Type thermocouple produces -2.223 mV at -50 °F and 29.515 mV at 1000 °F. This voltage is amplified by 300 so that it can be read by the ADC. The bias voltage is also introduced to ensure that the ADC only sees positive voltages. From Equations [10.9],[10.10], and [10.11], the number of milli-volts at the thermocouple is given by:

\[
\text{ConvGain}_{\text{mV/(count)}} = 6.008140
\]

\[
\text{ConvOffset}_{\text{counts}} = -409.5
\]

\[
\text{Thermocouple}_\text{mV} = \left( \frac{\text{ADC}_{\text{counts}}}{-409.5} \right) \times 0.008140
\]

Again, all you have to do is linearize the thermocouple based on the number of milli-volts read from the thermocouple. The pass counter for analog input channel #3 will also be set to 5 in order to read the thermocouple every 500 mS.
10.9.05 How to Use the Analog I/O Module, AX #4

Analog input channel #4 reads a voltage directly (maybe a battery). Because the voltage to read exceeds the FSV of the ADC, the hardware designer decided to simply divide the voltage in half. From Equations [10.9], [10.10], and [10.11], the voltage at the input is given by:

\[ \text{Gain}_{\text{V}}(\text{V/counts}) = 0.00488 \]

\[ \text{Offset}_{\text{counts}} = -0 \]

\[ \text{Voltage}_V = \left( \frac{\text{ADC}_{\text{counts}}}{2} - 0 \right) \times 0.00488 \]

The pass counter for analog input channel #4 will also be set to 10 in order to read the thermocouple every second.

10.9.06 How to Use the Analog I/O Module, AX #5

Analog input channel #5 reads a pressure from a pressure transducer which produces 2.6 mV/PSIG (pounds per square inch gauge). From Equations [10.9], [10.10], and [10.11], the pressure read by the transducer is given by:

\[ \text{Gain}_{\text{PSIG}}(\text{PSIG/counts}) = 0.009392 \]

\[ \text{Offset}_{\text{counts}} = 0 \]

\[ \text{Pressure}_{\text{PSIG}} = \left( \frac{\text{ADC}_{\text{counts}}}{2} - 0 \right) \times 0.009392 \]

The pass counter for analog input channel #5 will be set to 1. In order to read the pressure every 100 ms.

Let’s assume that the hardware designer came up with the circuit shown in Figure 10.18 to update the analog outputs.
10.09.07 How to Use the Analog I/O Module, AO #0

Analog output channel #0 is used to display temperatures from -50 °F to 200 °F on a 0 to 100 μA meter. A display of -50 °F is obtained with 55 DAC counts (0 μA) while 200 °F is obtained with 155 DAC counts (99.609 μA). The DAC counts are given by:

\[
\begin{align*}
\text{ConvGain} = & \quad (\text{counts})/(\text{°F}) = 1.02 \\
\text{ConvOffset} = & \quad 51 \\
\text{DAC counts} = & \quad 1.02 \times \text{Temperature} + 51
\end{align*}
\]

The pass counter for analog output channel #0 will be set to 1 in order to update the meter every 100 ms.
10.09.08 How to Use the Analog I/O Module, \(\text{\#1}\)

Analog output channel \#1 is used to control the opening of a valve. The valve is closed when the control current is 4 mA and wide-open when the control current is 20 mA. The counts vs. output current is given by:

\[
DAC_{\text{counts}} = \frac{2^n \times \text{Out}_{\text{mA}}}{\text{FSV} \times \left(\text{mA}/V\right)}
\]

A 12-bit DAC is used because a 10-bit DAC would not have the required resolution. Using a 10-bit DAC, 4 mA would require 205 counts (Equation [10.36]), while 20 mA would require 1023 counts, a range of 818 counts, or 0.122 percent. Note that 11-bit DACs are not commercially available. A 12-bit DAC requires 819.2 counts for a 4 mA output and 4095 counts for 20 mA (actually 19.995 mA). The DAC counts required to control the DAC are given by:

\[
\begin{align*}
\text{ConvGain}_{\text{ counts}/\%} & = \frac{4095 - 819.2}{100\%} = 32.758 \\
\text{ConvOffset}_{\text{ counts}} & = 819.2 \\
\text{DAC}_{\text{counts}} & = 32.758 \times \text{Input}_{\text{mA}} + 819.2
\end{align*}
\]

The pass counter for analog output channel \#1 will be set to 1 in order to update the valve every 100 mS.

10.09.09 How to Use the Analog I/O Module, \(\text{\#2}\)

Analog output channel \#2 is used to display the RPM of a rotating device on a 0 to 100 \(\mu\)A meter. A display of 0 RPM is obtained with 0 DAC counts (0 \(\mu\)A), while 6000 RPM is obtained with 1023 DAC counts (99.902 \(\mu\)A). The DAC counts are given by:

\[
\begin{align*}
\text{ConvGain}_{\text{ counts}/\text{RPM}} & = 0.1705 \\
\text{ConvOffset}_{\text{ counts}} & = 0 \\
\text{DAC}_{\text{counts}} & = 0.1705 \times \text{RPM} + 0
\end{align*}
\]

The pass counter for analog output channel \#2 will be set to 2 in order to update the meter every 200 mS.

The code to initialize the analog I/O channels is:


void AppInitADC (void) {

    ADCfgConv(0, 0.061050, 307.125, 10); /* Analog Inputs */
    ADCfgConv(1, 0.034886, 2293.2, 1);
    ADCfgConv(2, 0.061050, 409.3, 5);
    ADCfgConv(3, 0.009140, 409.3, 5);
    ADCfgConv(4, 0.004884, 0.0, 10);
    ADCfgConv(5, 0.009392, 0.0, 1);

    ADCfgScaling(1, /* Pointer to STD code */, /* Pointer to args */);
    ADCfgScaling(2, /* Pointer to TC code */, /* Pointer to args */);
    ADCfgScaling(3, /* Pointer to TC code */, /* Pointer to args */);

    ADCfgConv(0, 1.02, 51.0, 255, 1); /* Analog Outputs */
    ADCfgConv(1, 12.75, 819.2, 4096, 2);
    ADCfgConv(2, 0.1705, 8.0, 1023, 2);
}

You can now obtain the value read by any analog input channels by using ADCval() and set any analog output channel by calling ADSet().

10.10 Bibliography

Burns, G.W., Scrogg, M.G., Stoupe, G.E., Croackin, M.C. and Osehrie, W.F.  
Temperature-Electromotive Force Reference Functions and Tables for the Letter-Designated  
Thermocouple Types Based on the ITS-90 (NIST Monograph 175)  
United States Department of Commerce  
National Institute of Standards and Technology (NIST)  
Gaithersburg, MD 20899  
(801) 975-3068

Morgan, Don  
Numerical Methods, Real-Time and Embedded Systems Programming  
San Mateo, CA  
M&T Publishing, Inc.  

U.S. Software  
14215 NW Science Park Dr.  
Portland, OR 97229  
(503) 641-8446

Zoch, Eugene L.  
Data Acquisition and Conversion Handbook  
Mansfield, MA  
Data/Interface, 1979
Listing 10.1 AIO.C

/******************************************************************************
 * AIO Module
 * (C) Copyright 1999, Jean J. Labrosse, Weston, FL
 * All Rights Reserved
 *
 * Filename : AIO.C
 * Programmer : Jean J. Labrosse
 ******************************************************************************/

#define AIO_SOURCES 
#include "includes.h"

/******************************************************************************
 * LOCAL VARIABLES
 ******************************************************************************/

static unsigned AIO_stat = AIO_high; 
static unsigned AIO_ctrl; 

/******************************************************************************
 * LOCAL FUNCTION PROTOTYPES
 ******************************************************************************/

void AIO_Init(unsigned *stat);
static void AIO_Load();
static void AIO_Update(unsigned *val);

/*More*/
Listing 10.1 (continued) AIO.C

/*
 * DESCRIPTION: This function is used to configure an analog input channel.
 * Parameters:
 * gain: the gain for the calibration
 * offset: the calibration offset
 * Return:
 * 0 if successful
 * 1 if you specified an invalid analog input channel number.
 */

#define AININPUT (chipb0 n, pin2 gain, pin2 offset)

if (in < AIOINPUT_LIMIT)
{
  gpio = LATCH_PIN;
  gain = AININPUT(chipb0, n, pin2 gain);
  offset = AININPUT(chipb0, n, pin2 offset);
  return 0;
}
else
{
  return 0;
}
/* Stage */
Listing 10.1 (continued) AIO.C

/**
 * 
 * DESCRIPTION: This function is used to configure an analog input channel.
 * 
 * ARGUMENTS:
 * n: is the analog channel to configure (0..AIOxNCH-1).
 * gain: is the conversion gain
 * offset: is the conversion offset
 * 
 * RETURNS:
 * 0 if successfull;
 * 1 if you specified an invalid analog input channel number.
 */

INTNU AIOConfig(int n, FALSE gain, FALSE offset, INTNU pass)
{
    INTNU enc;
    AII "pain;"

    if (n < AIOxNCH)
    {
        enc = AAYRD[0]; /* Point to Analog Input structure */
        guaRead(AAUNAME, 0, enc); /* Obtain exclusive access to AI channel */
        encode=AIOxGain(); /* Store new conv. gain and offset into struct */
        pain=enc; /* pain=AIOxGain(); */
        pain=AIOxOffset; /* Compute overall offset */
        pain=AIOxOffset; /* pain=AIOxOffset + offset */
        pain=AIOxGain; /* pain=AIOxGain + gain */
        cGAINWrite(AIbest); /* Release AI channel */
    }
    else return (1);
}
/*GAIN*/
Listing 10.1 (continued) AIO.C

#define AIO_MAGIC 0x1badf00d

/*
 * DESCRIPTION: This function is used to configure the scaling parameters associated with an analog input channel.
 * Arguments:
 *  argc is the number of arguments passed to the program
 *  argv is a pointer to a null-terminated array of arguments
 *  sio is a pointer to a scaling function
 *  sio is a pointer to a scaling function
 * Returns:
 *  0 if successful,
 *  negative if invalid scaling input
 */

int main(int argc, char **argv)
{
    int i;

    if (argc < 2)
        return 0;

    /* Parse arguments */
    for (i = 1; i < argc; i++)
    {
        if (strcmp(argv[i], "-i") == 0)
        {
            sio = AIO_MAGIC;
            break;
        }
    }

    if (sio == -1)
    {
        printf("Invalid scaling input
");
        return 1;
    }

    /* Process input data */
    for (i = 2; i < argc; i++)
    {
        // Process each argument
    }

    return 0;
}

/* */
Listing 10.1 (continued) AX0.C

```c
/*
 * DESCRIPTION: This function is used to get the current value of an analog input channel in engineering units.
 * ARGUMENTS:
 *   c: Is the analog input channel (0..AX0.MAX_INPUT-1).
 *   val: Is a pointer to the destination engineering units of the analog input channel.
 * RETURNS:
 *   0 if successful.
 *   1 if you specified an invalid analog input channel number.
 *   In this case, the destination is not changed.
 */

int AX0Get(int c, double *val)
{
    double *phio;

    if (c < AX0.MAX_INPUT) {
        phio = (double *)Ax0Info[c].phio;
        AXEngine[AX0_INFO][c].get_out激情;
        *val = phio[0]; /* Obtain exclusive access to AI channel */
        AXEngine[AX0_INFO][c].set_in激情;
        /* Set the engineering units of the analog input channel */
        return 0;
    } else {
        printf("Invalid AI channel.
```
```
Listing 10.1 (continued) AIO.C

* ANA0G INPUTS INITIALIZATION *
* Description: This function initializes the analog input channels. *
* Arguments: None *
* Returns: None *
*
/

static void AINinit(void)
{
    int i;
    AIO *pAIO;

    pAIO = MATR1[0];
    for (i = 0; i < ADC_MAX_AIO; i++) {
        pAIO->AIno = AIN[0]; /* Analog channel is not bypassed */
        pAIO->AInew = 0x6000; /* New counts of ADC or DAC */
        pAIO->AInew = (PP2)1.0; /* Engineering units of AI channel */
        pAIO->AInew = (PP2)1.0; /* Total gain */
        pAIO->AInew = (PP2)1.0; /* Total offset */
        pAIO->AInew = 0; /* Pass count */
        pAIO->AInew = 1; /* Pass count */
        pAIO->AInew = (PP2)1.0; /* Calibration gain */
        pAIO->AInew = (PP2)1.0; /* Calibration offset */
        pAIO->AInew = (PP2)1.0; /* Conversion gain */
        pAIO->AInew = (PP2)1.0; /* Conversion offset */
        pAIO->AInew = (PP2)1.0; /* Input to scaling function */
        pAIO->AInew = (PP2)1.0; /* No function to execute */
        pAIO->AInew = (PP2)1.0; /* No arguments to scale section */
        pAIO++;
    }
*/

*/GINAE*/
Listing 10.1 (continued) AIO-C

void AIOInit(void)
{
   INIT err;

   xInit();
   AIOInit();
   AIOInit();
   AIOMem = OsemCreate(1); /* Create a mutual exclusion semaphore for AIOs */
   OSsemCreate(AIOMem, (void *)(AAL_MEM_AIOMEM, str_OFFSET), AIO_TASK_Prio);

   /* UNI*/

   /*
   ****************************************************/
   */

   /* AIO TASK */
   "DESCRIPTION : This task is created by AIOInit() and is responsible for enabling the analog input and
   output channels.
   * PARAMETERS : None.
   * RETURN : None.
   */

   void AIOCheck (void *data)
   {
      INIT err;

      data = data;
      /* Avoid compiler warning */
      OSsemEnter(AIOMem, 0, &err); /* Delay between execution of AIO manager */
      AIOMemInit(); /* Obtain exclusive access to AI channels */
      AIORead(); /* Update all AI channels */
      OsemEnter(AIOMem); /* Release AI channels (Allow high pri. task to run) */
      OsemEnter(AIOMem, 0, &err); /* Obtain exclusive access to AO channels */
      AOUpdate(); /* Update all AO channels */
      OsemEnter(AIOMem); /* Release AO channels (Allow high pri. task to run) */
   }

   /* UNI*/
Listing 10.1 (continued) AIO.C

/
******************************************************************************************
*
* Description: This function is used to set the engineering units of a bypassed analog input channel.
* This function is used to update the present of the sensor. This function is only valid if the bypass "Bypass" is open.
* Arguments:
* in is the analog input channel (0..AIO_MAX_AI-1).
* val is the value of the bypassed analog input channel.
* Returns:
* 0 if successful.
* 1 if you specified an invalid analog input channel number.
* 2 if AIThreshold was not set to "off".

******************************************************************************************
/

#define AIThreshold (25520 n. FFD3 val)

AIO *pAll;

if (in = AIO_MAX_AI { /* Faster to use a pointer to the structure */
    if (val = AIThreshold { /* Valid if the analog input channel is bypassed */
        pAll->AIChannels[in].
        ORL_16THRESHOLD();
        ORL_RESET_IORD));
    return (0);
    } else {
    return (1);
    }
} else {
    return (1);
}

/*END*/
/*
 * SET THE STATE OF THE BYPASS SWITCH
 *
 * Description: This function is used to set the state of the bypass switch. The analog input channel is bypassed when the ‘switch’ is open (i.e. AICBypassOn is set to 0). 
 *
 * Arguments:
 *   n: the analog input channel (0..AD10,AD11)
 *   state: the state of the bypass switch:
 *      0: DISABLES the bypass, i.e. the bypass ‘switch’ is closed
 *      0: ENABLES the bypass, i.e. the bypass ‘switch’ is open
 *
 * Returns: 0 if successful.
 *           1 if you specified an invalid analog input channel number.
 */

if (n < AD10,AD11) {
    AICBypassOn = state;
    return 0;
} else {
    return 1;
}

/*SOURCE*/
Listing 10.1 (continued) AIO.C

/**
 * UPDATE ALL ANALOG INPUT CHANNELS
 *
 * Description: This function processes all of the analog input channels.
 * Arguments:  None.
 * Returns:    None.
 */

static void AIOupdate (void)
{
   穴位 : AIO *pain

    pain = AAI[0];    /* Point at first analog input channel */
    for (i = 0; i < AAI_NUMOAL; i++) {    /* Process all analog input channels */
        if (pain->AAINumCh == i) {    /* See if analog input channel is bypassed */
            pain->AAINumCh = -1;    /* Decrement pass counter */
            if (pain->AAINumCh == 0) {    /* When pass counter reaches 0, read and scale AI */
                pain->AAINumCh = pain->AAINumCh + 1;    /* Reload pass counter */
                pain->AAINew = AAI[i];    /* Read AI for this channel */
                pain->AAINewLen = (parent->AAINew + pain->AIOOffset) * pain->AAINew;    /* Error */
                if (pain->AAINewLen == parent->AAINew) {    /* See if function defined */
                    (*pain->AAINewLen) (parent->AAINew);    /* Yes, execute function */
                    if (pain->AAINewLen == parent->AAINew) {    /* No, just copy data */
                        pain->AAOUT = pain->AAINewLen;    /* Output of scaling function to R.H. */
                    }
                    pain++;    /* Point at next AI channel */
                }
            }
        }
    }
}/*PLETED*/
Listing 10.1 (continued) AIO.C

/*
 * CONFIGURE THE CALIBRATION PARAMETERS OF AN ANALOG OUTPUT CHANNEL.
 *
 * Description: This function is used to configure an analog output channel.
 * Arguments:
 * in - the analog output channel (AXIOS0, AXIOS1, AXIOS2);
 * gain - the calibration gain (0..255);
 * offset - the calibration offset;
 * Returns:
 * 0 - if successful;
 * 1 - if you specified an invalid analog output channel number.
 */

#define AIOU(number, PP32 gain, PP32 offset)
{
    int n = AXIOS[number];
    /* Point to Analog Output structure */
    AXIOS[number].AXIOSlock = 0, share;
    /* Obtain exclusive access to AD channel */
    gain = gain + ADCON0GAIN;
    /* Store new cal. gain and offset into register */
    ADCON0 = offset;
    /* Compute overall gain */
    ADCON1 = offset + ADIAS0Offset + ADIAS1Offset;
    /* Compute overall offset */
    ADKon[AXIOS[number].AXIOSlock];
    /* Release AD channel */
    return (0);
}

/*END*/
Listing 10.1 (continued) AIO.C

/*
  \* Description: This function is used to configure an analog output channel.
  \* Arguments: n is the analog channel to configure (0..AIO_MAX_AO-1).
  \* gain is the conversion gain.
  \* offset is the conversion offset.
  \* pass is the value for the pass counts
  \* Returns: 0 if successful.
  \* 1 if you specified an (invalid analog output channel) number.
  \*/

DEFRU_AIOConf (DEFRU n, FP32 gain, FP32 offset, INT32 len, INT32 pass)
{

  DEFRU err;
  AIO *pio;

  if (n < AIO_MAX_AO) {
    pio = AIO(n);
    /* Point to AIO channel object */
    /* Obtain exclusive access to AO channel */
    pio->AIOConf = gain;
    /* Store new conv. gain and offset. Use struct */
    pio->AIOConf.offset = offset;
    pio->AIOConf.len = len;
    pio->AIOConf.pass = pass;
    /* Release AO channel */
    return 0;
  } else 
    return (1);
}

/*END*/
Listing 10.1 (continued) AIO.C

/*
 * CONFIGURE THE SCALING PARAMETERS OF AN ANALOG OUTPUT CHANNEL.
 *
 * Description: This function is used to configure the scaling parameters associated with an analog output channel.
 *
 * Arguments: n: Is the analog output channel to configure (0..AIO_MAX_OUTPUT-1).
 *            arg: Is a pointer to arguments needed by the scaling function.
 *            fact: Is a pointer to a scaling function.
 *
 * Returns: 0, if successful.
 *          -1, if you specified an invalid analog output channel "n".
 */

#elif defined(BI06)  
void (*AIO_scaling)(void *fact, void *arg)
{
    if (n > AIO_MAX_OUTPUT) {  
        / * Pointer to a pointer to the structure */
        return (0);
    } else {
        return (1);
    }
} /*AIO_scaling*/
Listing 10.1 (continued) AIO.C

```c
/*
 * AIO.C - Analog Input Output Channel Definitions
 *
 * DESCRIPTION: This function initializes the analog output channels.
 * ARGUMENTS: Init: void
 * RETURNS: None
 *
 */

static void AIOInit(void);

// Definitions

#define AIO_INPUTS 5
#define AIO_OUTPUTS 2

// Function prototypes

void AIOInit(void);

// Variables

// AIO inputs

// AIO outputs

// Function definitions

void AIOInit(void)
{
    // Initialize AIO inputs
    // Initialize AIO outputs
}
```

Listing 10.1 (continued) ATO.C

/*
  * DESCRIPTION: This function is used to set the OUTPUT value of an analog output channel.
  * Arguments:
  * chan   Is the analog output channel (0..AUXMUX_A0-1).
  * value  Is the desired analog output value in Engineering units.
  * If you specified an invalid analog output channel number, return (-1).
*/

#include <stdio.h, aux.h, util.h>

AOSET (int chan, int value)
{
    if (0 <= chan <= AUXMUX_A0-1)
        AUXMUX(chan) = value;  /* Set the engineering units of the analog output channel */
    return (0);
    } else {
    return (-1);
    }

/* SYNTAX */
Listing 10.1 (continued) AIO.C

/**
 * @brief Set the state of the bypassed analog output channel.
 * @param n is the analog output channel (0..AIO_MAX-1).
 * @param val is the value of the bypassed analog output channel.
 * @returns
 * 0 if successful,
 * 1 if you specified an invalid analog output channel number,
 * 2 if AIO_isBypass is not on.
 *
 */

#include "AIO.h"

uint AIOsetBypass(uint n, VPI2 val)
{
    AIO *pAIO;

    if (n < AIO_MAX)
    {
        pAIO = &AIO[n];
        if (pAIO->AIO_isBypass) /* See if the analog output channel is bypassed */
            pAIO->AIO_bypassed = val;
        else
            /* Bypass to use a pointer to the structure */
            return (3);
    } else /* return (2); */
    return (1);

} /* AIOsetBypass */
Listing 10.1 (continued) AIO.C

/*
 * GET THE STATE OF THE BYPASS SWITCH
 */

/* Description: This function is used to set the state of the bypass switch. The analog output channel is bypassed when the switch is open (i.e. AIO_Bypass is set to TAKE).
 * Arguments: n is the analog output channel (0..AIO_OUTPUT-1)
 * state is the state of the bypass switch
 *             TRUE enables the bypass (i.e. the bypass switch is closed)
 *             FALSE disables the bypass (i.e. the bypass switch is open)
 * Returns: 0 if successful
 *          1 if you specified an invalid analog output channel number.
 */

int AIOGetBypassState (int n, BOOL state)
{
    int err;

    if (n < AIO_OUTPUT) {
        AIO_Bypass[n] = state;
        return (0);
    } else {
        return (1);
    }
}

/* end */
/*
 * Description: This function processes all of the analog output channels.
 * Arguments: None.
 * Returns: None.
 */

static void ANOPrep(void)
{
    uint16_t *pio = &ANOPin[0];
    for (i = 0; i < ANOPins; i++)
    {
        if (pio->ANOPin = -1) /* See if analog output channel is bypassed */
            pio->ANOPin = pio->ANOPin + 1;
        if (pim->ANOPin = pio->ANOPin + 1) /* Point at first analog output channel */
            pio->ANOPin = pio->ANOPin + 1;
        if (pim->ANOPin = -1) /* Process all analog output channels */
            pio->ANOPin = pio->ANOPin + 1;
    }
}

/* GPIO */
Listing 10.1 (continued) A0.C

/*
 * DESCRIPTION: The function is called by A0OnOff() to initialize the physical I/O used by the A0
 * 
 * Arguments: none.
 * 
 * Returns: none.
 */

void A0OnOff (void)
{
    /* This is where you will need to put your initialization code for the ADCs and DACs.
     * You should also consider initializing the contents of your DMA to a known value.
     */
}

/*
 * DESCRIPTION: This function is called to read a physical ADC channel. The function is supposed to
 * also control a multiplexer if there are multiple analog input lines connected to the ADC.
 * 
 * Arguments: ch is the ADC logical channel number (0-8), myDevice.A0[0...7].
 * Returns: The raw A0-000000 from the physical device.
 */

void Read_ADC (DMA) of
{
    /* This is where you will need to provide the code to read your ADC(s).
     * */
    /* 
     * Note: If the ADCs are used in a different way, you will have to convert this logical channel
     * number to the actual physical port locations (or addresses) where your ADL and AD1s are located.
     */
    /* 
     * This is responsible for:
     * 1) Selecting the proper MUX channel.
     */
    /* 
     * 2) Waiting for the MUX to stabilize.
     */
    /* 
     * 3) Reading the data from the ADC and
     */
    /* 
     * 4) Waiting for the ADC to complete the conversion.
     */
    /* 
     * 5) Reading the counts from the ADC and
     */
    /* 
     * 6) Returning the counts to the calling function.
     */

t1等候

/* */
Listing 10.1 (continued) AIO.C

*"*******************************************************************************/
/*@ Description: This function is called to write the 'raw' counts to the proper analog output device */
/*@           : (e.g. DAC). It is up to this function to direct the DAC (through the proper DAC if more */
/*@ than one DAC is used) */
/*@ Arguments: ch (the DAC logical channel number (0..25,MAX,10-1)). */
/*@ Code: see the DAC code to write to the DAC */
/*@ Returns: */
*******************************************************************************/
*/
void AOWfrey_chr(int ch, int16_t outs)
{
    int ch = ch;
    outs = outs;
    /*This is where you will need to provide the code to update your DAC(s). */
    /* Add () is passed a "DAC החץ" channel number. You will have to connect this logical channel */
    /* number into actual physical port locations (or addressess) where your DACs are located. */
    /* Map ( ) is responsible for writing the counts to the selected DAC based on a logical number. */
}

/*endf */
Listing 10.2 AIO.H

/*
  ***********************************************************************************************
  *  Analog I/O Module
  *
  *  (C) Copyright 1999, Jean J. Labrosse. Mention, 4U
  *  All Rights Reserved
  *
  *  Filename:   AIO.H
  *  Programmer: Jean J. Labrosse
  ***********************************************************************************************
  */

#ifdef AIO_DEFINES
#define AIO_DEF
#undef
#endif AIO_DEF extern
#endif

  /***********************************************************************************************
  *  CONFIGURATION CONPSANTS
  ***********************************************************************************************
  */

#ifndef CNRL_H
#define AIO_MAX_IN 80
#define AIO_MAX_OUT 100
#define AIO_MAX_INP 512

#define AIO_MAX_AT 8 /* Maximum number of Analog Input Channels (1..255) */
#define AIO_MAX_AC 8 /* Maximum number of Analog Output Channels (1..255) */
#endif

/*SPD*/
Listing 10.2 (continued) AIO.H

* *

#define struct ai (* 

IOEnable IOEnable;
IOError IOError;
* Bits enable switch (0 means when TRUE)
* Engineering units of AI channel
* Total gain (AIDGainHi * AIDGainLo)
* Total offset (A IID OffsetLo + AIDOffsetHi)
* Maximum count of an analog output channel
* Pass counts
* Pass counter (loaded from PassCtrl)
* Calibration gain
* Calibration offset
* Conversion gain
* Conversion offset
* Output to scaling function
* Output from scaling function
* Function to execute for further processing
* Pointer to argument to pass to AIDoffSet.

void (*AIOSetScaleOffset)(input *point); /*/global VARIABLES*/

/AIO
*/

/AIO_DATA AIO;
/AIO_DATA AIOalnum;
/AIO_DATA AIO_data;

/*GINS*/
Listing 10.2 (continued)  AIO-H

void ADD(sincos);  /* 2. hardware dependent functions */
Asynchronous
Serial Communications

The world of data communications is very complex. A single book (let alone a chapter) cannot cover everything. Data communication is concerned specifically with the issues that must be considered when communicating data between two devices (generally computers). When computing elements are distant from one another, in most cases data is transmitted serially. Because data in a computer is handled in parallel (8 bits or more), it is necessary to convert this information from parallel to serial (when sending) and from serial to parallel (when receiving). There are basically three modes of communication, as shown in Figure 11.1:

1. Simplex: Data travels in one direction (from A to B). An example of a simplex link would be scoreboard displays such as those used in hockey, basketball, or other sports. The information is entered at a console by the score/timer and sent serially to large displays that everybody can see.
2. Half-duplex: Data travels in one direction (from A to B) and then the other direction (from B to A) but not at the same time. The RS-485 interface (discussion starts on page 408) is half-duplex.
3. Full-duplex: Data can travel in both directions at the same time.
In this chapter, I will briefly discuss asynchronous communications, the RS-232C standard, the RS-485 standard, the serial ports on a PC, and how data is sent and received on an asynchronous communication port. This chapter is not concerned with what is actually sent and received. In other words, in this chapter, I will not cover data communication protocols. This chapter provides three software modules:

1. A low-level driver that allows characters to be sent and received on either of the two serial I/O ports on a PC. The driver is called COMP_PC and is interrupt-driven.
2. An interface to the low-level driver (described previously) which allows bytes sent and received to be buffered. This interface allows you to use buffered serial I/O without requiring a real-time operating system. This software module is called COMP_MIO and is applicable to just about any foreground/background system.
3. An interface to the low-level driver which assumes the presence of a real-time operating system. This software module (called COMP_RTS) allows you to use buffered serial I/O in a multitasking environment.

The code provided in this chapter doesn’t make any assumption about the communication mode, i.e., simplex, half-duplex, or full-duplex.

11.00 Asynchronous Communications

You can find just about everything there is to know about asynchronous serial communications in the excellent book by Joe Campbell, *C Programmer’s Guide to Serial Communications*, which is now in its second edition (see “Bibliography” on page 455). If you are further interested in the world of data communications, you should also add the books from Andrew S. Tanenbaum and Fred Halsall to your collection.

In asynchronous communication systems, the receiver clock is not synchronized to the transmitter clock when data is being transmitted between two devices. Generally speaking, asynchronous transmission
is used to indicate that data is being transmitted as individual bytes. Each byte is preceded by a start signal and terminated by one or more stop signals. The start and stop signals are used by the receiver for synchronization purposes. As shown in Figure 11.2, the transmission line is in a mark (binary 1) condition in its idle state. As each byte is transmitted, it is preceded by a start bit which is a transition from a mark to a space (binary 0). This transition indicates to the receiving device that a byte is being transmitted. The receiving device detects the start bit and the data bits that make up the byte. At the end of the byte transmission, the line is returned to a mark condition by one or more stop bits). At this point, the transmitter is ready to send the next byte. The start and stop bits permit the receiving device to synchronize itself to the transmitter on a byte-by-byte basis. From Figure 11.2, you should note that bytes are transmitted least-significant bit first. Also, each byte of data being transmitted requires at least two bits which are used for synchronization purposes. The synchronization bits thus impose an overhead of 20 percent.

**Figure 11.2** Asynchronous communications timing diagram.

![Asynchronous communications timing diagram](image)

If it is assumed that the receiver knows how fast each bit is being transmitted, the transmission rate is known as the baud rate. As long as the sender and the receiver agree to use the same baud rate, the actual rate used is not important. The industry has, however, standardized baud rates, as shown in Table 11.1.

### Table 11.1 Standard baud rates

<table>
<thead>
<tr>
<th>Baud rate</th>
<th>Bit time (μs)</th>
<th>#Bytes/sec. (note 1)</th>
<th>Time between bytes (μs) (note 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>3.3333</td>
<td>30</td>
<td>33.333</td>
</tr>
<tr>
<td>1200</td>
<td>1666.6</td>
<td>60</td>
<td>16.667</td>
</tr>
<tr>
<td>2400</td>
<td>3333.33</td>
<td>30</td>
<td>4.167</td>
</tr>
<tr>
<td>4800</td>
<td>6666.67</td>
<td>45</td>
<td>2.083</td>
</tr>
<tr>
<td>9600</td>
<td>104.19</td>
<td>960</td>
<td>1.042</td>
</tr>
<tr>
<td>19200</td>
<td>51.25</td>
<td>1920</td>
<td>521</td>
</tr>
<tr>
<td>38400</td>
<td>25.63</td>
<td>3840</td>
<td>240</td>
</tr>
<tr>
<td>56000</td>
<td>17.91</td>
<td>5600</td>
<td>179</td>
</tr>
</tbody>
</table>

*Note 1: Assuming 1 start, 8 data bits, and 1 stop.*
Asynchronous communications is performed almost transparently by a device called a UART (Universal Asynchronous Receiver Transmitter). To send and receive data, your program simply writes and reads bytes to and from the UART. UARTs are generally capable of sending and receiving data at the same time (i.e., they support full-duplex communication). A UART appears to the microprocessor as one or more memory locations or I/O ports. UARTs generally contain one or more status registers, which are used to verify the progress and state of data transmission and reception. The microprocessor can thus know when a byte has been received, whether a communication error occurred, or when a byte has been sent. UARTs can also be configured through one or more control registers. Configuration of a UART consists of setting the baud rate, setting the number of stop bits (1, 1-1/2 or 2), enabling interrupts when bytes are sent or received, etc.

Probably the most popular UART is the National Semiconductor NS16550 (see 1.4.5.50.pdf on the companion CD-ROM). There are many other UARTs available on the market and some of the more popular ones are: the AMD Z8530, the Motorola 6800 ACIA, the Zilog Z840 SIO, etc. The NS16550 contains all the required functionality to send and receive characters, but the NS16550 also is equipped with an internal Baud Rate Generator, which makes it especially easy to interface to most microprocessors. What is nice about UARTs is that they also are available on a large number of single-chip CPUs. Embedded systems can thus benefit from the capability of communicating with terminals, computers or even other embedded microprocessors.

Data sent and received by UARTs can consist of anything that can be represented by eight bits (or less) or any multiple of eight bits. You can thus send binary data, ASCII (American Standard Code for Information Interchange) characters, BCD (Binary Coded Decimal) digits, etc. By far the most important character set used by the English-speaking world is ASCII. ASCII is a 7-bit code. The mapping of a 7-bit binary value to an ASCII code is shown in Figure 11.3. ASCII characters are used to represent strings in C. For example, the string "HELLO" is represented by the following ASCII codes:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>O</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0x48</td>
<td>0x45</td>
<td>0x4C</td>
<td>0x4F</td>
<td>0x00</td>
</tr>
</tbody>
</table>

The ASCII chart contains two columns of "special" characters. Some of these ASCII characters are well known to C programmers: NL (Null character, 0x00), BS (Bell, 0x07), BS (Back Space, 0x08), LF (Line Feed, 0x0A), CR (Carriage Return, 0x0D), FF (Form Feed, 0x0C), ESC (Escape, 0x1B), and SP (Space, 0x20). The first two columns also contain character codes that can be used in data communication protocols (beyond the scope of this book).
### Figure 11.3  ASCII character set (7-bit code).

<table>
<thead>
<tr>
<th>LSD</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B30L</td>
<td>BLE</td>
<td>SP</td>
<td>D</td>
<td>@</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td></td>
<td></td>
<td>A</td>
<td>G</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>STX</td>
<td>02</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>ETX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>BS</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>LF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>VT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>FF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>CR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>SO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 11.01 RS-232C

Dating all the way back to 1969, the RS-232C standard is probably the most widely used communication interface in the world. RS-232C was defined by the Electronic Industries Association (EIA) and is formally known as: "Interface between data terminal equipment and data communication equipment employing serial binary data interchange." As shown in Figure 11.4, the RS-232C standard is a hardware protocol used to interface between two devices: one is called the Data Terminal Equipment (DTE) and the other, the Data Communication Equipment (DCE). The RS-232C standard defines:

1. The mechanical aspects of the interface.
2. The characteristics of the electrical signals.
3. The functional aspects of the interchange.
The RS-232C standard says that there should be two 25-pin connectors: the male connector is used on the DTE while the female connector is used on the DCE. The actual type of connector is not defined by the standard. The industry has, however, standardized on 25 pins D-shell type connectors.

Electrically speaking, the RS-232C standard specifies that:

- the load capacitance on a driver is not to exceed 2500 picofarads (pF),
- the load resistance on a driver must be between 3000 and 7000 ohms,
- the data signaling rate (or baud rate) must be below 20,000 bits per second (bps) under the specified load,
- the maximum levels on the RS-232C lines are not to exceed 15 volts (with respect to signal ground),
- drivers must be able to produce between ±5 and ±15 volts (logic 1) and -5 to -15 volts (logic 0),
- inputs must be able to accept signals from ±3 to ±15 volts (logic 1) and -3 to -15 volts (logic 0).

Under the maximum load suggested by the RS-232C standard, the distance between the DTE and the DCE should not exceed 50 feet. Simple math would have you conclude that at a distance of 25 feet (half the capacitance) you should be able to increase the signaling rate to 40,000 bps, 80,000 bps at 12.5 feet, and about 160,000 bps at 2 feet. In fact, many communication packages allow you to interface two computers at a data signaling rate of up to 115,200 bps. You should note that the RS-232C standard does not define "standard" baud rates. The RS-232C standard allows data to be sent and received at the same time (i.e., full-duplex).

From the 25 pins defined by the RS-232C standard only nine (9) lines are actually used in "real-world" applications. Probably for that reason and to reduce cost, IBM started to use 9-pin connectors for RS-232C communication when they introduced the IBM PC/AT back in the mid-1980s. The nine pins that are retained for RS-232C communications are shown in Table 11.2. You should note that communication ports on PCs are generally connected as DTEs (i.e., male connectors).
Table 11.2 RS-232C connections.

<table>
<thead>
<tr>
<th>Description</th>
<th>Acronym</th>
<th>DTE DB-25M Pin#</th>
<th>DTE DB-9M Pin#</th>
<th>Direction</th>
<th>DCE DB-9F Pin#</th>
<th>DCE DB-25F Pin#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmit</td>
<td>TxD</td>
<td>3</td>
<td>3</td>
<td>&lt;</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Receive Data</td>
<td>RxD</td>
<td>2</td>
<td>2</td>
<td>&lt;</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Request To Send</td>
<td>RTS</td>
<td>7</td>
<td>7</td>
<td>&gt;</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Clear To Send</td>
<td>CTS</td>
<td>8</td>
<td>8</td>
<td>&lt;</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Data Set Ready</td>
<td>DSR</td>
<td>6</td>
<td>6</td>
<td>&lt;</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>Data Carrier Detect</td>
<td>DCD</td>
<td>1</td>
<td>1</td>
<td>&lt;</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Data Terminal Ready</td>
<td>DTR</td>
<td>4</td>
<td>4</td>
<td>&lt;</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Ring Indicator</td>
<td>RI</td>
<td>9</td>
<td>9</td>
<td>&lt;</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>Signal Ground</td>
<td>SG</td>
<td>5</td>
<td>5</td>
<td>&lt;</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

A full description of the use of each of the pins is beyond the scope of this chapter because the code presented in this chapter only assumes the presence of the TxD, RxD, and RTS lines. You will find, however, detailed information about these lines in Joe Campbell’s book.

An RS-232C communications port generally consists of a UART and what are called EIA drivers/receivers. The EIA drivers and receivers are used to convert microprocessor levels (typically 0 to 5 volts) to RS-232C compatible levels: –3 to –15 volts (logic 0) to +3 to +15 volts (logic 1). An RS-232C DTE using an NS-16550 and EIA drivers/receivers is shown in Figure 11.5. Inverters are used for electrical reasons. For your convenience, Figure 11.5 shows the pinout for both the DB25 and DB9 connectors. (Note that the "M" in DB-25M and DB-9M stands for "Male." Only one of the two connectors, however, would actually be used.)

**Figure 11.5 RS-232C connections (DTE).**
Connection between a DTE and a DCE is quite straightforward and is shown in Figure 11.6. A readily available DB25F to DB25M (or DB9F to DB9M) cable is typically all that is required.

**Figure 11.6 RS-232C connections (DTE to DCE).**

There might be situations where you would need to connect two DTEs together. For example, you may want to connect a terminal to a PC or even interface two PCs. Connecting two DTEs together is a little tricky because:
- Both DTEs have male connectors and,
- outputs would be connected to outputs and, inputs would be connected to inputs on each DTE.

This situation can be solved by using what is called a Null Modem adapter (also known as a Gender Changer) or by using two female connectors and making the connections shown in Figure 11.7.

**Figure 11.7 RS-232C NULL Model (DTE to DTE).**
Communication between DTEs is also possible by using only three wires as shown in Figure 11.8. The unused inputs must be asserted to satisfy the UART (specifically, the TxD output line typically is disabled when CTS is negated). This can be accomplished by asserting the DTR output on each DTE.

The software modules presented in this chapter assume that you are using a three-wire interface.

**Figure 11.8 RS-232C 3-wire DTE to DTE.**

**11.02 RS-485**

The RS-232C standard requires that a direct connection be made between two devices. This is known as a point-to-point interface. If, for example, you need to communicate with many embedded microprocessors, you would need to dedicate an RS-232C port for each embedded processor, as shown in Figure 11.9. This situation can become expensive if the embedded processors are located far from the PC. A16, RS-232C is fairly susceptible to noise because of its common ground arrangement.
The RS-485 interface has been created to allow multiple (up to 32) processors to communicate with each other on a common line. RS-485 is sometimes called a party-line or a multi-drop interface and is shown in Figure 11.10. The RS-485 interface uses differential line driver/receiver chips (such as the Texas Instruments SN75176A Differential Bus Transceiver) and only requires a single twisted pair of wires. Communication on an RS-485 interface is, however, half-duplex. Each communicating element on an RS-485 interface is called a node and communication generally follows a MASTER/SLAVE protocol (but doesn’t have to). One of the nodes is called the MASTER while all other nodes are called SLAVES. In a MASTER/SLAVE arrangement, all communication occurs between the MASTER and a SLAVE (not between SLAVES). Each node on an RS-485 is assigned a unique node ID number. Node 0 is generally assigned to the MASTER. The MASTER selectively communicates with one of the SLAVES at any given time. An RS-485 interface has the following features:

- very noise immune,
- maximum cable length of 4000 feet,
- data signaling rate up to 10 Mbps (megabits per second),
- capable of supporting up to 32 nodes, and
- capable of supporting a multi-MASTER configuration.
Communication on an RS-485 interface proceeds as shown in Figure 11.11. The MASTER enables its transmit line driver and sends a command or data to a SLAVE (2). The desired SLAVE I.D. number is typically sent in the first byte in the message from the MASTER. When all bytes of the command or data are sent, the MASTER enables its transmit line driver (2) and waits for a reply from the SLAVE. The SLAVE processes the command or data received and formulates a response for the MASTER (5). The SLAVE enables its transmit line driver (6) and sends the response back to the MASTER. When all bytes which make up the response from the SLAVE are sent, the SLAVE enables its transmit line driver (6). The MASTER analyzes the response from the SLAVE (8) and performs whatever action is needed. The MASTER is then ready to initiate the next command or data transfer. You should note that when either the MASTER or the SLAVE is sending data the respective receivers are monitoring what is being sent. The data sent can be verified by the sender to ensure the integrity of the line, or the receiver can simply discard the same number of bytes received as sent. The sender can also ignore any received data until it is done with the transmission.
The NS16550 is not a good UART to use for RS-485 communication because it doesn’t provide an interrupt when the last byte has been transmitted. Instead, the NS16550 only tells you when it is ready to send another byte. Figure 11.12(a) will help illustrate what happens. The NS16550 contains two registers for data transmission: a Transmitter Holding Register (THR) and a Transmitter Shift Register (TSR). When you write a data byte to the NS16550, the byte is actually deposited into the TSR (0) and is then automatically transferred to the TSR (2). At this point, the bits in the TSR are shifted out at the baud rate that you selected (8) and an interrupt is generated by the NS16550 to indicate that theTHR can accept another byte (9); the THR holds the byte while the previous byte is being transmitted. If you disable the RS-485 line driver in the THR Interrupt Service Routine (ISR), you will actually prevent the next byte from being sent because it is still in the process of being shifted out.


Figure 11.12 Disabling the RS-485 line driver.

What you actually need is a UART that interrupts the processor when the STOP bit of the last byte has been shifted out, as illustrated in Figure 11.12(b). In this case, there is no need for a THR. The CPU writes a byte to the TSR (11), which then gets shifted out by the UART (12). When the start bit, the byte, and the stop bit are sent, the UART interrupts the CPU (13). If there are no more bytes to send, the ISR disables the line driver (14).

The low-level code provided in this chapter is designed to work with the NS16550 and so it does not support RS-485. It should, however, be fairly easy to port the code to another UART which supports the scheme described in Figure 11.12(b).

11.03 Sending and Receiving Data

As previously mentioned, data is sent and received by a UART by writing and reading from memory or I/O port locations. A bit in the UART’s status register can be monitored to determine when a byte has been received. Similarly, another bit can be examined to see when a byte has been transmitted through the interface. This method of monitoring the UART status is called polling the I/O device and generally is used when the microprocessor can monitor the status register faster than bytes are sent and received. Polling has serious shortcomings, especially for input, because bytes can be missed while the processor is occupied with other duties. Because microprocessors have other things to do besides wait for serial I/O ports, it is common to resort to an interrupts-driven scheme to handle data reception and transmission.
11.03.01 Receiving Data

When using an interrupt-driven scheme, an interrupt is generated when a byte arrives through the serial port. The interrupt handler reads the byte from the port, which generally clears the interrupt source. At this point you have the choice of either processing the byte received in the ISR or putting the byte into some sort of buffer to let a background process handle the data. When you use a buffer, the size of the buffer depends on how quickly your background process can get control of the CPU to process the information. For example, if the worst case latency of your background process is 200 ms, you should plan for a buffer of at least 192 bytes if your serial port receives bytes at 9600 baud (960 bytes/sec × 200 ms). A special type of buffer called a ring buffer (also called a circular buffer) is often used to capture data from a serial port.

To avoid allocating very large buffers, you can resort to what is called flow control. Basically, the interrupt receiving data can notify the sender that the receiver’s buffer is getting full. The sender would then hold off with its transmission until the receiver empties out the buffer and notifies the sender that it can proceed. The most common flow control scheme is called XON-XOFF and it uses the ASCII characters DC1 (0x11) for XON (i.e., “send me more”) and DC3 (0x13) for XOFF (i.e., “don’t send me any more”). Using the XON-XOFF scheme precludes you from sending binary data because the data you are sending could happen to be one of these two characters.

Flow control can also be performed by using some of the RS-232C lines. This would allow you to send and receive binary data. Unfortunately, the RS-232C standard doesn’t specify which lines to use when you are not interfacing to a modem. Nothing prevents you from using the modem control lines RTS, CTS, DSR, and DTR, but you will have to establish how flow control will work between your devices.

Input buffering using a ring buffer is shown in Figure 11.3. When bytes are received, the ISR reads the byte from the serial port (1) and places the byte into the ring buffer (2). Your application code (background) then monitors the ring buffer to see if bytes have been received (3). If the ring buffer is not empty, the “oldest” byte (least recent byte) is extracted from the ring buffer.

Figure 11.13 Buffered I/O, receiving bytes.

The following pseudocode for both the ISR and the interface function to your application follow. Actual code for the ISR and the interface function will be described later.
ISR (ReceiveISR) (void)
{
    INTR c;
    Save processor context;
    c = Get byte from RX port;
    if (RX Ring buffer not full) {
        Put byte received into ring buffer;
    }
    Restore processor context;
    return from interrupt;
}

ISR (CompleteISR) (void)
{
    INTR c;
    c = NOL;
    Disable interrupts; /* Prevent ISRs during access */
    if (RX Ring buffer not empty) {
        c = Get byte from ring buffer;
    }
    Enable interrupts;
    return (c);
}

You should note that interrupts are disabled when your application accesses the ring buffer to ensure exclusive access to the ring buffer from either the ISR or the interface function. If your application doesn't extract bytes from the ring buffer in time, the ring buffer will become full and received bytes will be lost.

The response to incoming data depends on how soon your background process gets to execute. If you are using a real-time kernel, you can process incoming data almost as quickly as you receive it without doing so in an ISR. To accomplish this, a semaphore is added to the management of the ring buffer as shown in Figure 11.4. In this case, your application waits on the semaphore (S). When a byte is received, the ISR reads the byte from the serial port (S) and deposits it in the ring buffer (S). The ISR then signals the semaphore to indicate to the waiting task that a byte was received (E). Signaling the semaphore makes the waiting task ready to run. When the ISR completes, the kernel determines if your waiting task is now the highest-priority task ready to get the CPU. If it is, the ISR resumes the task waiting for the byte (assuming a preemptive kernel). Your application code then extracts the byte from the ring buffer and performs whatever processing is required.
Figure 11.14 Buffered serial I/O with semaphore, receiving bytes.

The following pseudocode for both the ISR and the interface function to your application follow. Actual code for the ISR and the interface function will be described later. As with the previous scheme, if your application doesn’t extract bytes from the ring buffer in time, the ring buffer will become full and bytes received will be lost. The use of a real-time kernel, however, reduces the chance of this situation from happening.

Most real-time kernels allow you to specify the maximum amount of time your task is willing to wait for a byte to be received. This gives your task a chance to take corrective action in case something happened to the communication link. For example, a task can send a message and then wait for a response. If the response doesn’t arrive within a certain amount of time, the sender can conclude either that there is nobody listening or that something happened to the transmission medium.
ISR ComRxISR (void)
{
    INBPU c;

    Save processor context;
    Tell OS that we are processing an ISR;
    c = Get byte from RX port;
    if [Rx Ring Buffer is not Full] {
        Put received byte into Ring Buffer;
        Signal Rx Semaphore;
    } 
    Tell OS that we are exiting an ISR;
    Restore processor context;
    Return from Interrupt;
}

INBUU ComGetChar (INBUU *err)
{
    INBUU c;

    Wait for byte to be received (using semaphore with T.O.);
    if [time out] {
        *err = Time out error;
        return (0);
    }
    Disable interrupts;
    c = Get byte from Ring Buffer;
    Enable interrupts;
    *err = No error;
    return (c);
}

Signalling the semaphore everytime a character is received can consume valuable CPU time. An alternate method is to only signal the semaphore when a special character is received. For example, you can signal the semaphore when a carriage return character (i.e., CR or 0x0D) is received. You application can then be notified once a full command is received which reduces the overhead. Of course, your buffer needs to have sufficient storage to hold one or more commands. This alternate method is shown in the following pseudocode.
ISR CommISR (void)
{
    INIEMU c;
    
    Save processor context;
    Tell OS that we are processing an ISR;
    c = Get byte from RX port;
    if (Rx Ring Buffer is not Full) {
        Put received byte into Ring Buffer;
        if (received byte is the end-of-command byte) {
            Signal Rx Semaphore;
        }
    }
    Notify OS that we are exiting an ISR;
    Restore processor context;
    Return from Interrupt;
}

ISR CommGetCommand (INIEMU *command, INIEMU *bytes)
{
    INIEMU c;
    INIEMU nrx;
    Wait for command to be received (using semaphore with T.O.);
    if (timed out) {
        *bytes = 0;
        return (Timeout error);
    }
    nrx = 0; /* Clear number of bytes received counter */
    Disable interrupts;
    c = Get byte from Ring Buffer;
    while (*c != end-of-command byte) {
        nrx++;
        /* Clear number of bytes received counter */
        c = Get byte from Ring Buffer;
    }
    Enable interrupts;
    *bytes = nrx error; /* Get number of bytes received */
    return (No error);
}
11.03/02 Transmitting Data

Transmission of bytes works somewhat like byte reception. Your background process deposits bytes in an output buffer. When the transmitter on the UART is ready to send a byte, an interrupt is generated, the byte is extracted from the buffer, and the ISR outputs the byte. There is, however, one small complication: The serial port generates an interrupt only AFTER the port has finished sending the byte. The most elegant way I found to resolve this dilemma is to disable interrupts from the transmitter until you need to send bytes. Interrupts are enabled AFTER the output buffer is loaded with at least one byte. As soon as you allow the transmitter to interrupt, the first byte to send will be removed by the transmit ISR and output to the UART. The ISR then examines the buffer and, if there are no more bytes to send, the ISR disables the transmit interrupt.

Buffering of data makes a lot of sense when you have to transmit a relatively large amount of data on the serial port, such as the contents of a disk file. Output buffering uses a ring buffer as shown in Figure 11.15. When one or more bytes need to be sent, they are placed in the ring buffer (1). Transmit interrupts are enabled after putting a byte into the buffer (2). If the UART is ready to send a byte, an interrupt occurs and the ISR extracts the "oldest" (least recent) byte from the ring buffer (3). The byte is then output to the serial port (4). Transmit interrupts will be inhibited if the byte extracted from the buffer makes the ring buffer empty.

Figure 11.15 Buffered serial I/O, transmitting bytes.

The following pseudocode for both the ISR and the interface function to your application follows. Actual code for the ISR and the interface function will be described later.
void CommBuffer (INT8U c)
{
    Disable interrupts; /* Prevent INTs during access */
    if (Tx Ring Buffer is not Full) {
        Put byte to send into ring buffer;
        if (This is the first byte in the ring buffer) {
            Enable tx interrupts;
        }
    }
    Enable interrupts; /* Allow CPU interruptions */
}

ISR CommCharISR (void)
{
    
    Save processor context;
    if (Tx Ring Buffer not empty) {
        c = Get next byte to send from ring buffer;
        Output byte 'c' to TX port;
    } else {
        Disable tx interrupts;
    }
    Restore processor context;
    Return from interrupt;
}

Figure 11.16 shows how you can make use of a real-time kernel’s facilities. The semaphore is used as a traffic light passing the sending task when the ring buffer is full. To send data, the task waits for the semaphore (5). If the ring buffer is not full, the task proceeds to deposit the byte into the ring buffer (6). Transmitter interrupts are enabled if the byte deposited is the first byte in the ring buffer (3). The transmit interrupt ISR extracts the “oldest” byte from the ring buffer (8) and signals the semaphore (8) to indicate that the ring buffer has room to accept another character. The ISR then outputs the byte to the UART.
Figure 11.16 Buffered serial I/O with semaphore, transmitting bytes.

It is important to note that the semaphore needs to be a counting semaphore, and the semaphore must be initialized to the size of the ring buffer. The pseudocode for both the interface function to your application and the ISR follows. Actual code for the ISR and the interface function will be described later.

```c
void Comistream(int c, int *err)
{
    wait for space in the Tx Ring Buffer using semaphore T O I;
    if (closed out)
    {
        *err = Time out error;
        return;
    }
    Enable interrupts;
    Put byte to send (c) into the Tx Ring Buffer;
    if (this is the first byte in the Tx Ring Buffer)
    {
        Enable TX interrupts;
    }
    Enable interrupts;
    *err = No error;
}
```
IIR ComWriteCharIn (void)
{
    INTUR c;

    Save processor context;
    if (Tx Ring Buffer is not empty) {
        c = Get next character to send from Tx Ring Buffer;
        Output character ‘c’ to Tx port;
        Signal Tx semaphore;
    } else {
        Disable Tx Interrupts;
    }
    Restore processor context;
    Return from Interrupt;
}

11.04 Serial Ports on a PC

The software modules provided in this chapter allow you to use both serial ports on an IBM-PC/AT compatible computer although it can be easily altered to support different hardware. A review of the PC’s architecture relating to the serial ports available on PCs is thus necessary in order to better understand the code.

PCs are typically equipped with two RS-232C communication ports that are referred to as COM1 and COM2. Both ports generally consist of a National Semiconductor NS16550 or equivalent UART and are capable of communicating at baud rates up to 115200 bps. The PC provides services through its BIOS (Basic Input/Output System) but unfortunately, communications using the BIOS must be done by polling (monitoring the port to see if bytes have been received or sent). This limitation means that communication effectively cannot exceed about 1200 baud. This shortcoming can be corrected by replacing the BIOS services with interrupt-driven functions.

An IBM-PC/AT computer contains two interrupt controllers (Intel 82C59A PIC) providing 15 sources of interrupts to the PC’s microprocessor. Interrupts are labeled IRQ0 through IRQ15, as shown in Figure 11.17. IRQ2 of the first 82C59A is actually the output of the second 82C59A interrupt controller.
Table 11.3 shows what devices are typically connected to the interrupt controllers. The table lists the interrupt sources in priority order (IRQ0 has the highest priority). Table 11.3 also shows that each serial I/O port is connected to its own IRQ line: COM1 is connected to IRQ4 while COM2 is connected to IRQ3.
Table 11.3 PC/AT interrupts summary.

<table>
<thead>
<tr>
<th>IRQ#</th>
<th>Description</th>
<th>Interrupt vector #</th>
<th>Interrupt vector address</th>
<th>Mask register address</th>
<th>Mask bit#</th>
<th>Clear IRQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x09</td>
<td>Timer (i.e., tick, 18.2 Hz)</td>
<td>0x08</td>
<td>0x0100:0x0020</td>
<td>0x0021</td>
<td>0</td>
<td>0x020</td>
</tr>
<tr>
<td>0x0A</td>
<td>Keyboard</td>
<td>0x09</td>
<td>0x0000:0x0044</td>
<td>0x0021</td>
<td>1</td>
<td>0x020</td>
</tr>
</tbody>
</table>
| 0x0B | (Async-8-15
  shows, below)         | 0x0A               | 0x0000:0x0049            | 0x0021                | 2         | 0x020     |
| 0x0D | Real-Time Clock             | 0x07               | 0x0000:0x0137            | 0x00A1                | 0         | 0x0A0 then 0x0C0 |
| 0x0E | Keyboard in IRQ9            | 0x13               | 0x0100:0x0100            | 0x00A1                | 1         | 0x0A0 then 0x020 |
| 0x0F | Unassigned                   | 0x72               | 0x0000:0x001C             | 0x00A1                | 2         | 0x0A0 then 0x020 |
| 0x0F | Unassigned                   | 0x73               | 0x0000:0x010C              | 0x00A1                | 3         | 0x0A0 then 0x020 |
| 0x10 | Unassigned                   | 0x74               | 0x0000:0x010D            | 0x00A1                | 4         | 0x0A0 then 0x020 |
| 0x11 | 8087 co-processor            | 0x75               | 0x0100:0x01A4            | 0x00A1                | 5         | 0x0A0 then 0x020 |
| 0x12 | Hard Disk                    | 0x76               | 0x0000:0x0A1B            | 0x00A1                | 6         | 0x0A0 then 0x020 |
| 0x13 | Unassigned                   | 0x77               | 0x0000:0x01DC            | 0x00A1                | 7         | 0x0A0 then 0x020 |
| 0x14 | COM2                         | 0x08               | 0x0000:0x0022            | 0x0021                | 1         | 0x0000     |
| 0x15 | COM1                         | 0x0C               | 0x0000:0x0030            | 0x0021                | 4         | 0x0000     |
| 0x16 | LPT2                         | 0x0F               | 0x0000:0x0044            | 0x0021                | 5         | 0x0020     |
| 0x17 | Floppy Disk                  | 0x0E               | 0x0000:0x007F            | 0x0021                | 6         | 0x0020     |
| 0x18 | LPT1                         | 0x0F               | 0x0000:0x0032            | 0x0021                | 7         | 0x0020     |

IRQ4 is asserted whenever a byte is either received on COM1 or whenever COM1 has completed the transmission of a byte. When an interrupt occurs, the CPU automatically vectors to the Interrupt Vector Address shown in Table 11.3. The Interrupt Vector Address points to the Interrupt Service Routine (ISR) responsible for handling the source of the interrupt: either a byte was received, a byte was sent, or both. IRQ3 works just like IRQ4 except that it uses a different vector.

As shown in Figure 11.1B, COM port interrupts have to travel through many "doors" (gates) in order to actually interrupt the CPU. First, interrupts must be asserted by the CPU by setting the I/F bit in the PSTW (Processor Status Word). Second, the interrupt controller can inhibit interrupts from any device connected to it through the 8259A Interrupt Mask Register. Finally, the NS16550 UART is capable of inhibiting either the Rx (byte received) or the Tx (byte sent) interrupts through its Interrupt Enable Register.
11.05 Low-Level PC Serial I/O Module (COMM_PC)

This section describes a driver that I wrote which makes much better use of the serial I/O ports provided on a PC. The code and the functionality of the driver easily can be ported to other environments. Your application actually interfaces with two modules, as shown in Figure 11.19. Note that the term PC is used generically to mean any PC having either an Intel 80286, 80386, 80486, or Pentium microprocessor.

The low-level driver is responsible for interfacing with the National Semiconductor NS16550 UART. Functions are provided to your application to configure the two ports (COM1 or COM2), enable/disable communication interrupts, and acquire/release the COM port interrupt vectors. The interface functions will be described later.

Your application also interfaces to either one of two buffered serial I/O modules: COMM32ND or COMM686. You would use COMM32ND in a foreground/background application and COMM686 if you are running a real-time kernel such as VOGIC-8.

This section specifically describes the low-level driver interface functions. The source code for the low-level code is found in the \SOFTWARE\BLOCKS\COMMSRC directory, and specifically, in the following files:

- COMM_PCI.ASM (Listing 11.1)
- COMM_PCI.C (Listing 11.2)
- COMM_PCI.H (Listing 11.3)
As a convention, all functions and variables related to the low-level serial I/O module start with Comb while all MFC functions start with CMH.

CombISR() and CombISR1() (COM_PC.ASM) are the functions that are executed when an interrupt occurs on the PC’s COM1 or COM2, respectively. These functions start by saving the CPU registers onto the current task stack or the background stack in a foreground/background system. If you are using C8051F020, CombISR() needs to increment the µCOS-I global variable OSIntNext() after saving the CPU registers and call OSIntExit() prior to restoring the registers. After incrementing OSIntNext(), the ISR calls CombISRHandler().

CombISRHandler() is responsible for doing most of the ISR processing and knows about the NS16550 UART interface. You can easily expand this function to support more than just two serial ports. CombISRHandler() determines whether the interrupt was caused by the reception of a byte, the completion of a byte transmission, or both.

If a byte is received, CombISRHandler() reads the UART’s receive data register and calls CombPutChar() CombPutChar() (described later) is a function that knows what to do with the byte just received. In our case, the byte received is placed in a ring buffer.

If the interrupt is caused by the completion of byte transmission, CombISRHandler() calls CombGetChar() (described later) to see if anything else needs to be sent. When all bytes have been sent, CombISRHandler() disables further transmit interrupts from the UART. The interrupt source is not cleared because CombISRHandler() does not actually write to the UART’s transmit data register (there is nothing to send). The next time your application code puts something in the ring buffer the transmit interrupt will be re-enabled and an interrupt will occur immediately. The ISR will then extract the byte to send from the ring buffer and satisfy the UART.

Before returning to CombISR() or CombISR1(), CombISRHandler() clears the interrupt from the PC’s IEC59A interrupt controller.
CommCfgPort() is used to establish the characteristics of a serial port. You will need to call this function before calling any of the other services provided by this module for the specific port.

Arguments

- **ch** specifies the channel and can be either COM1 (for the PC's COM1) or COM2 (for the PC's COM2).
- **baud** specifies the desired baud rate. The NS16550 sets the baud rate (i.e., baud) according to the following equation:
  \[ \text{baud rate, divisor} = \frac{115200}{\text{baud}} \]
  You can specify just about any baud rate except that the baud rate divisor will be truncated to a 16-bit integer. For example, you can specify 7500 baud, but you will actually get 7680, as shown:
  \[ \frac{115200}{7500} = 15.36 \]
  Truncation produces a baud rate divisor of 15 and the NS16550 UART will actually be set to a baud rate of 115200 / 15 = 7680.
- **bits** specifies the number of bits used. The NS16550 supports either 5, 6, 7, or 8. Generally, you would specify 7 bits with either ODD or EVEN parity or 8 bits with NO parity.
- **parity** specifies the type of parity checking used by the serial port. You can specify either:
  - **COMM_PARITY_NONE** for no parity
  - **COMM_PARITY_ODD** for odd parity
  - **COMM_PARITY_EVEN** for even parity
- **stop*/ specifies the number of stop bits used. The NS16550 supports either 1 or 2. You would typically specify 1 stop bit, though.

Return Value

CommCfgPort() returns either **COMM_NO_ERR** (if the channel you specified was either COM1 or COM2) or **COMM_COMM_RST**.

Notes/Warnings

In the previous edition of this book, CommCfgPort() only allowed you to configure the baud rate. The number of bits was always assumed to be 8, the parity was always set to NONE, and the number of stop bits 1.
Example

void main (void)
{
    INIT err;
    
    ConfigPort(COM1, 9600, 8, COMN_PARITY_NONE, 1);
    
}

CommRxFlush()

void CommRxFlush(int ch);
(COMM_PC.C)

CommRxFlush() allows your application to clear the contents of the UART's receive register. The receive register on the NS16550 UART can receive a byte while another byte waits for the CPU to be processed. CommRxFlush() simply discards the last received byte. If you use the more powerful NS16550 UART then you would set COMM_MUX_PCK (in COMM_PC.H or CFG.C) to 16 because this chip can buffer up to 16 characters.

Arguments

ch specifies the channel and can be either COM1 (for the PC's COM1) or COM2 (for the PC's COM2).

Return Value

None

Notes/Warnings

None

Example

The following code example assumes the presence of anRTOS but the function can just as easily be used in a foreground/background environment.

void Task (void *pdata)
{

    for (i = 0; i < 10; i++)
        CommRxFlush(COM1);

    for (i = 0; i < 50; i++)
        CommRxFlush(COM2);

    for (i = 0; i < 20; i++)
        CommRxFlush(COM3);

}


CommRxIntDis() is used to prevent interrupts from the desired serial port when bytes are received. 
CommRxIntDis() hides the details of disabling interrupts for the selected serial port from your application. Note that CommRxIntDis() will ensure that the interrupt controller bit will not be cleared (disabling the port’s interrupts) if the UART’s transmit interrupt is enabled.

Arguments

eh specifies the channel and can be either COMM (for the PC’s COM1) or COMM2 (for the PC’s COM2).

Return Value

None

Notes/Warnings

None

Example

The following code example assumes the presence of an RTOS but the function can just as easily be used in a foreground/background environment.

void task (void *pdata)
{
    
    for (;;) {
        CommRxIntDis(COMM2);
        
    }
}
CommRxIntEn() is used to enable interrupts from the desired serial port when bytes are received. CommRxIntEn() hides the details of enabling interrupts for the selected serial port from your application. Enabling interrupts consist of setting bit 0 of the UART’s Interrupt Enable Register (IER) and clearing the appropriate bit on the PC’s 8255A interrupt controller.

**Arguments**

dn specifies the channel and can be either COM1 (for the PC’s COM1) or COM2 (for the PC’s COM2).

**Return Value**

None

**Notes/Warnings**

None

**Example**

The following code example assumes the presence of an RTOS but the function can just as easily be used in a foreground/background environment.

```c
void Task (void *pdata)
{

    for (;;) {
        CommRxIntEn(COM1);
        .
        .
    }
}
```
CommTxIntDis() is used to prevent interrupts from the desired serial port when bytes are sent. CommTxIntDis() hides the details of disabling interrupts for the selected serial port from your application. Note that CommTxIntDis() will ensure that the interrupt controller bit will not be cleared (disabling the port’s interrupts) if the UART’s receive interrupt is enabled.

Arguments

ch specifies the channel and can be either COM1 (for the PC’s COM1) or COM2 (for the PC’s COM2).

Return Value

None

Notes/Warnings

None

Example

The following code example assumes the presence of an RTOS but the function can just as easily be used in a foreground/background environment.

```c
void TxCallback (void *pParam)
{
    
    for (;;) {
        CommTxIntDis (COM1);
        
    }   
}
```
CommTxIntEn()

void CommTxIntEn(uint8_t ch);

(compu_pc.c)

CommTxIntEn() is used to enable interrupts when a byte is sent by the UART. CommTxIntEn() hides the details of enabling interrupts for the selected serial port from your application. Enabling transmission interrupts consist of setting bit 1 of the UART’s Interrupt Enable Register (IER) and clearing the appropriate bit on the PC’s 8259A interrupt controller.

Arguments

ch specifies the channel and can be either COM1 (for the PC’s COM1) or COM2 (for the PC’s COM2).

Return Value

None

Notes/Warnings

None

Example

The following code example assumes the presence of an RTOS but the function can just as easily be used in a foreground/background environment.

void Task (void *pdata)
{
    
    for (;;) {
        
        CommTxIntEn(COM2);
        
    }
}
CommSetIntVect() is used to set the contents of the Interrupt Vector Table (IVT) for the desired serial port (see Table 11.3). CommSetIntVect() saves the old contents of the IVT (i.e., a pointer to the BIOS communication handler) so that it can be recovered when your application code returns to DOS.

Arguments

ch is the serial channel to process and can either be COM1 or COM2. When you specify COM1, CommSetIntVect() places a pointer to CommISR() at address 0x0000:0x0030 (see Table 11.3). Similarly, when you specify COM2, CommSetIntVect() places a pointer to CommISR() at address 0x0000:0x002C (see Table 11.3).

Return Value

None

Notes/Warnings

None

Example

```c
void main(void)
{
    int err;
    
    CommSetPort(COM1, 5600, COM_ENABLE_MODEM, 8, 0);
    CommSetIntVect(COM1);
    CommSetPort(COM1);
}
```
CommRclIntVect() is used to restore the original interrupt vectors of the desired serial port in the IVT (Interrupt Vector Table).

Arguments

ch is the serial channel to process and can either be COM1 or COM2. When you specify COM1, CommRclIntVect() places the previous vector for the PC's COM1 at address 0x0000:0x0020 (see Table 11.3). Similarly, when you specify COM2, CommRclIntVect() places the previous vector for the PC's COM2 at address 0x0000:0x0020 (see Table 11.3).

Return Value

None

Notes/Warnings

None

Example

The following code example assumes the presence of an RTOS but the function can just as easily be used in a foreground/background environment.

    void Task (void *pdata)
    {
        ...
        for (;;) {
            ...
            if (done with serial port #1 and returning to DOQ) {
                CommRclIntVect(COM1);
                ...
            }
        }
    }
11.06 Buffered Serial I/O Module (COMMGBKND)

The COMMGBKND module allows data received from and sent to a UART to be buffered. Specifically, you would use the COMMGBKND module if you write an application destined for a foreground/background environment. The COMMGBKND module is designed to work in conjunction with the COMM_PC module described in the previous section. COMMGBKND allows you to do full-duplex communication on either serial port concurrently. The source code for the COMMGBKND module is found in the SOURCES/BLOCKS/COMMGBKND directory and specifically in COMMGBKND.C (Listing 11.4) and COMMGBKND.H (Listing 11.5).

WARNING
In the previous edition of this book, COMMGBKND was called COMMSBUF. The file COMMSBUF.C is now COMMGBKND.C and, COMMSBUF.H is now COMMGBKND.H.

As a convention, all functions and variables related to the COMMGBKND module start with Comm while all define constants start with COMM.

Each serial port is assigned two ring buffers: one for byte reception and another for byte transmission. Both ring buffers are stored in a structure called COMM_RING_BUF (see COMMGBKND.C on page 473). Each ring buffer consists of four elements:

1. a storage for data (an array of INTELS)
2. a counter containing the number of bytes in the ring buffer
3. a pointer where the next byte will be placed in the ring buffer
4. a pointer where the next byte will be extracted from the ring buffer

Figure 11.20 shows a flow diagram for data reception using the COMMGBKND module and how it interfaces with the COMM_PC module. CommBuff?? are elements of the COMM_RING_BUF data structure. An interrupt occurs when a byte is received by the UART (Q). If interrupts are enabled, the CPU vector to the appropriate ISR, i.e., CommISR(COMMISKIP) saves the CPU's context (its registers), and calls CommISRHandler(1) (Q). CommISRHandler() gets the byte from the UART and calls CommPutBufChar() in order to save the byte into the ring buffer (Q). Reading the byte from the UART clears the interrupt from the UART. If the buffer is not already full, a counter, which keeps track of how many bytes are in the buffer is incremented. (CommBuffCnt) Next, the byte retrieved from the UART is stored at the location pointed to by CommBufInPtr (Q). The pointer is then incremented and checked to make sure that it still points somewhere in CommBuf[]. If CommBufInPtr points past the array, it is re-initialized to point at CommBuf[0].
Figure 11.20 Buffered serial I/O, receiving bytes.

Your application code can find out whether there are bytes in the ring buffer by calling CommIsEmpty() when data is available, it is extracted from the ring buffer by calling CommReceive(). If the buffer is not already full, a counter is incremented for each byte in the buffer. You could be sending to the serial port into the ring buffer by calling CommPutChar(). The pointer is incremented and checked to make sure that it still points somewhere in the ring buffer. If the buffer is not already full, it is re-initialized to point at the beginning of the array, i.e., RingBufPtr(0). When the transmit interrupt is enabled (2), the CPU then vectors to the appropriate ISR (CommISR()), saves the CPU's context, and calls CommISRHandler(). CommISRHandler() gets the byte from the ring buffer by calling CommGetChar(). Note that CommGetChar() obtains the byte from a different pointer than CommPutChar(). This allows the byte to be sent in the same order as they were placed in the buffer (First In First Out, FIFO). Obviously, when a byte is removed from the buffer, the byte count is decremented. Writing a byte to the UART clears the
interrupt, however, when there are no more bytes to send, `CommISBHandler()` will not write anything to the UART. Instead, further interrupts from the UART will be disabled. In this case, the interrupt status of the UART will remain active but will not make it to the processor until UART interrupts are again enabled.

**Figure 11.21 Buffered serial I/O, transmitting bytes.**

`CommGetChar()` is an interface function between the `COMISGND` module and the `COM_PC` module. The `COM_PC` module calls this function when a byte has been sent by the UART. Basically, this function says, "Give me the next byte to send." `CommGetChar()` returns the next byte to send from the transmit ring buffer if there is at least one byte in the ring buffer. If the buffer is empty, `CommGetChar()` returns the NULL character and tells the caller that there is no more data in the buffer. This allows the caller to disable further transmit interrupts until there is more data to send.
Chapter 11: Asynchronous Serial Communications — 437

CommGetChar()

INCLUDE CommGetChar (INTBU ch, INTBU *err);
(COMMAND.C)

CommGetChar() allows your application to extract data from the received data ring buffer.

Arguments

ch is the serial channel and can be either COM1 or COM2.

err is a pointer to a variable that will hold status about the outcome of the function. CommGetChar() sets *err to one of the following:

COMM_NO_ERR if a byte is available from the ring buffer.
COMM_RX_EMPTY if the ring buffer is empty.
COMM_BAD_CH if you do not specify either COM1 or COM2.

Return Value

The function returns the oldest byte stored in the ring buffer if the buffer is not empty. If the buffer is empty, CommGetChar() returns the NULL character (i.e., 0x00).

Notes/Warnings

None

Example

void ReadChar (void)
{
    INTBU err;

    .

    c = CommGetChar (COM1, &err);
    if (err == COMM_NO_ERR) {
        Process character;
    }
    .
}


CommInit()

```c
void CommInit(void);
(COMMONS.RD.C)
```

CommInit() is used to initialize the COMMONS module. This function must be called before any other services provided by this module. CommInit() clears the number of bytes in the ring buffer counter and also initializes both the IN and OUT pointers of each ring buffer to point at the beginning of the data storage area.

Arguments
None

Return Value
None

Notes/Warnings
None

Example
```c
void main (void)
{
    ...
    CommInit();
    ...
}
```
CommIsEmpty()

DECLARE CommIsEmpty(IN OUT ch);
(CommISRIO.C)

CommIsEmpty() allows your application to determine if a byte was received on the serial port.

Arguments
ch is the serial channel and can be either COM1 or COM2.

Return Value
The function returns TRUE if no data was received and FALSE if data is available in the ring buffer.

Notes/Warnings
If you specify an incorrect channel number the function returns TRUE to prevent you from extracting data from an invalid serial port.

Example

void SyncPort (void)
{
    DWORD err;
    ...
    if (!CommIsEmpty (COM1)) {
        /* Characters have been received */
    }
    ...
}
CommIsFull() allows your application code to check the status of the transmit ring buffer.

**Arguments**

ch is the serial channel and can be either COMM1 or COMM2.

**Return Value**

The function returns TRUE when the buffer is full and FALSE otherwise.

**Notes/Warnings**

If you specify an incorrect channel number, the function returns TRUE to prevent you from sending data to an invalid serial port.

**Example**

```c
void SendData (void)
{
    int err;

    // ...

    if (CommIsFull(COMM1)) {
        /* Characters can be sent to serial port */
    }
    // ...
}
```
Chapter 11: Asynchronous Serial Communications — 441

CommPutChar()

CommPutChar() allows your application to send data to a serial port (one byte at a time).

Arguments

ch is the serial channel and can be either COM1 or COM2.
c is the byte that your application sends to the serial port. The byte can have any value between 0x00 and 0xFF (i.e., you can send binary data).

Return Value

CommPutChar() returns a value representing the outcome of the function call as follows:

- **COMM_No.ERR** if the byte was placed in the ring buffer and will eventually be sent by the UART if a free space is available from the ring buffer.
- **COMM_BAD.CH** if you do not specify either COM1 or COM2.
- **COMM_TX_FULL** indicates that you tried to send a byte to an already-full buffer.

Notes/Warnings

If you configured the serial port to 7 data bits then you will not be able to send binary data.

Example

```c
char Message[] = "Hello World!";

void SendPort (void)
{
    ... Thunder ERR;
    ...
    err = COMM_NO.ERR;
    s = Message[0];
    while (*s &amp; err == COMM_NO.ERR) {
        err = CommPutChar(COM1, *s++);
        ... Thunder ...
    }
    ...
```
11.07 Buffered Serial I/O Module (COMRTOS)

The COMRTOS module works just like the COMMFD module except that the COMRTOS module uses semaphores to indicate when bytes are inserted into the buffer. Semaphores allow your task-level code to process incoming and outgoing data as quickly as possible. Furthermore, your application code no longer needs to poll the receive buffer to see if bytes are available. Similarly, your application code also will be suspended if the transmit buffer is full. This also prevents your code from having to check that the transmit buffer is not full when you are sending data on a serial port.

The source code for the COMRTOS module is found in the \SOFTWARE\BLOCKS\COMM\SOURCE directory and, specifically, in COMRTOS.c (Listing 11.6) and COMRTOS.h (Listing 11.7). As a convention, all functions and variables related to the COMRTOS module start with Comm while all #define constants start with Comm.

**WARNING**

In the previous edition of this book, COMMSKD was called COMMFD. The file COMMFD.C is now COMRTOS.C and, COMMFD.H is now COMRTOS.H.

Along with the two ring buffers, each serial port now has two semaphores: one to signal that a byte was received and the other to signal that a byte was sent. The COMM_RING_BSP structure (see COMRTOS.C on page 484) is identical to the COMMFD structure except for the addition of the semaphores.

**Figure 11.22 Buffered serial I/O, receiving bytes.**
Figure 11.22 shows a flow diagram for data reception using the COMMSOS module and how COMMSOS interfaces with the COMM_PC module. Your application will call CommGetChar() except that your task will be suspended if the buffer is empty. You can specify to CommGetChar() a time-out value so prevent suspending your application task forever. When a byte is received, your task will “wake-up” and will receive the byte from the serial port.

CommPutChar() is an interface function between the COMMSOS module and the COMM_PC module. The COMM_PC module calls this function when a byte is received. CommPutChar() deposits the byte into the receive ring buffer but only if the buffer is not already full. The byte is discarded if the buffer is full. When the byte is inserted in the buffer, CommPutChar() signals the data reception semaphore to indicate any pending task that data was received.

To prevent suspending your application code, you can find out whether there are bytes in the ring buffer by calling CommIsEmpty().

Figure 11.23 shows a flow diagram for data transmission using the COMMSOS module and how it interfaces with the COMM_PC module. Again, everything is identical to the COMMSOS module except for the semaphore. When you want to send data to a serial port, CommPutChar() waits for the semaphore. Because the transmit semaphore is initialized to the size of the buffer when the COMMSOS module is initialized, CommPutChar() will suspend your application code when there is no more room in the buffer. The suspended task will resume as soon as the UART catches up sending the bytes.

**Figure 11.23 Buffered serial I/O, transmitting bytes.**
returns the NUL character and tells the caller that there is no more data in the buffer. This allows the caller to
disable further transmit interrupts until there is more data to send. The data transmit semaphore is signaled
when a byte is extracted from the buffer. This indicates to the sending task that there is more room in the
transmit buffer.
CommGetChar() allows your application to extract data from the received data ring buffer.

**Arguments**

*ch* is the serial channel and can be either COM1 or COM2.

*to* specifies a timeout (in "clock ticks"). If a byte is not received on the serial port within this time, CommGetChar() will return to your application. Your task will wait for a byte forever when you specify a timeout of 0.

*err* is a pointer to a variable that will hold status about the outcome of the function. CommGetChar() sets *err* to one of the following:

- **COMM_NO_ERR** if a byte is available from the ring buffer within the timeout period.
- **COMM_RX_TIMEOUT** if no data is received within the specified timeout.
- **COMM_BAD_CH** if you do not specify either COM1 or COM2.

**Return Value**

The function returns the oldest byte stored in the ring buffer if the buffer is not empty. If the function times out, CommGetChar() returns the EOF character (i.e., 0x0D).

**Notes/Warnings**

None
Example

```c
void Task (void *data)
{
    DWORD err;

    for (rs) {
        c = CONGetChar (YMM, 0, &err);
        if (err == CON_NO_MSG) {
            Process character;
        }
    }
}
```
CommInit() is used to initialize the Comm module. This function must be called before any other services provided by this module. CommInit() clears the number of bytes in the ring buffer counter and also initializes both the IN and OUT pointers of each ring buffer to point at the beginning of the data storage area. The data reception semaphore is initialized to 0, indicating that there is no data in the ring buffer. The data transmission semaphore is initialized with the size of the transmit buffer, indicating that the buffer is empty.

**Arguments**
None

**Return Value**
None

**Notes/Warnings**
None

**Example**

```c
void main (void) {
   
   CommInit();

   
}
```
CommIsEmpty() allows your application to determine if a byte was received on the serial port. This function allows you to avoid task suspension if no data is available.

Arguments

ch is the serial channel and can be either COM0 or COM1.

Return Value

The function returns TRUE if no data was received and FALSE if data is available in the ring buffer.

Notes/Warnings

If you specify an incorrect channel number, the function returns TRUE to prevent you from calling CommGetChar() thinking that data is available from an invalid port.

Example

```c
void Task (void *pdata)
{
    INT8U err;
    
    for (;;) {
        
        if (CommIsEmpty(COM1) == FALSE) {
            c = CommGetChar(COM1, &err); /* Character available */
            
            
            Process character;
        }
        
        
    }
```
**CommIsFull()**

```c
BOOLEAN CommIsFull(DINTU ch);
```

CommIsFull() allows your application code to check the status of the transmit ring buffer. This function allows you to avoid task suspension if the buffer is already full.

**Arguments**

*ch* is the serial channel and can be either COM1 or COM2.

**Return Value**

The function returns TRUE when the buffer is full and FALSE otherwise.

**Notes/Warnings**

If you specify an incorrect channel number, the function returns TRUE to prevent you from calling CommPutChar() thinking that data can be sent to the serial port.

**Example**

```c
void Task (void *pParam)
{
    DINTU err;
    char *p;

    for (i = 0; i < 0x10000; i++) {
        if (CommIsFull(COM1) == FALSE) {
            err = CommPutChar(COM1, '\p', 0);
        }
    }
}
```
CommPutChar() allows your application to send data to a serial port (one byte at a time). CommPutChar() suspends the calling task if the transmit ring buffer is full. CommPutChar() will resume when a byte is removed from the ring buffer by the transmit ISR.

Arguments

ch is the serial channel and can be either COM1 or COM2.
e is the byte that your application sends to the serial port. The byte sent can have any value between 0x00 and 0xFF (i.e., you can send binary data).
to specifies the amount of time (in “clock ticks”) that CommPutChar() will wait for the buffer to clear up. If a byte is not transmitted on the serial port within this time, CommPutChar() will return to your application. Your task will wait forever when you specify a timeout of 0.

Return Value

CommPutChar() returns a value representing the outcome of the function call as follows:

COMM_NO_ERR  the byte was placed in the ring buffer and will be sent by the UART if a byte is available from the ring buffer.
COMM_BAD_CH  if you do not specify either COM1 or COM2.
COMM_TX_TIMEOUT indicates that the buffer didn’t clear up within the allowed time.

Notes/Warnings

If you configured the serial port to 7 data bits then you will not be able to send binary data.
Example

```c
char Message[] = "Hello World!";

void test (void)
{
    int err;
    char *s;

    for (; ; ) {
        s = Message;
        err = COM_SET_ESR;
        while (*s && err == COM_SET_ESR) {
            err = ComPortChar(COM0, *s++, 0);
        }
    }
}
```
11.08 Configuration

Configuration of the communications driver is very simple because all you have to do is change a few #defines to accommodate your environment.

COMM_PC.H (or CFG.H):

- COMM1_BASE and COMM2_BASE are the base port addresses for the PC’s COM1 and COM2. In most cases, you will not have to change these.
- COMM_MAX_RX sets the number of bytes that the UART buffers internally. For the NS16550 UART, you should set this constant to 16 because the NS16550 can be receiving a byte while another byte is waiting to be processed by the CPU.
- COMM мероприятии COMM2.H (or CFG.H):
  - COMM_RX_BUF_SIZE sets the size of the receive ring buffer for both serial ports. The size of the receive buffer can be as large as 65534 bytes.
  - COMM_TX_BUF_SIZE sets the size of the transmit ring buffer for both serial ports. As with the receive ring buffer, the size can be as large as 65534 bytes.

11.09 How to use the COMM_PC and the COMM_BGND Module

If you write a foreground/background application you will need to use the COMM_PC (assuming you are using a PC) and the COMM_BGND modules. The first thing you need to do is to configure the module by setting the value of the #defines described in section 11.08. Next, you will need to call functions to initialize the modules and the serial port(s) that you are planning on using. For example, if you are using the PC’s COM1, you would need to have the following code:

```c
void main(void)
{
    ...
    CommInit(); /* Initialize COMM_BGND */
    ...
    CommClosePort(COMM1, 9600, 0, COMM_PARITY_NONE, 1); /* Install the interrupt vector */
    CommEnableInterrupt(COMM1); /* Enable receive interrupts */
    ...
}
```
Chapter 11: Asynchronous Serial Communications — 453

You should note that you don’t need to enable transmit interrupts because this is done automatically
when you send data on the serial port. When all your initialization is done, your background loop could
check for incoming data, as shown.

```c
void main(void)
{
   DINT c;
   INT16U err;
   /* Initialization code described above -----------------------------*/

   while (1) { /* Background loop (infinite loop) */

      if (!CommEmpty(COM1)) {
         c = CommGetChar(COM1, &err);
         if (err == COMM_NO_STR) {
            /* Process received data ---------------------------------*/
            CommPutChar(COM1, ???); /* Send response */
         } else {
            /* Process communications error ------------------------*/
         }
      }
   }
}
```

11.10 How to use the COMM_PC and the COMMRTOS Module

If you write an application using a real-time kernel you will need to use the COMM_PC (assuming you are
using a PC) and the COMMRTOS modules. Again, the first thing you need to do is to configure the module
by setting the value of the #define described in section 11.08. Your startup code will need to create
the task(s) that will be responsible for servicing the serial port(s). You should have one task for each
serial port. The following segment of code is used to create the task that will handle COM1. You should
consult TEST.C (see Chapter 11) to see what else you need to properly initialize μC/OS-II.
#define COMM_TASK_PRIO 20 /* Define the priority of the task */

GL_STK CommStack(512);

void main(void)
{
    OSInit(); /* Initialize the O.S. (uc/OS-II) */
    
    OSTaskCreate(CommTask, &CommTaskStk(511), COMM_TASK_PRIO);
    
    OStart();
}

You should initialize the serial communications code from within the task that will handle the port(s). Using the PC's COM1, you would have the following code:

void CommTask(void *pdata)
{
    INMRU cr;
    INMRU err;
    CommInit(); /* Initialize COM1/RUGS */
    CommConfigure(COM1, 9600, 8, COM8_PARITY_NONE, 1);
    CommSetIntVect(COM1); /* Install the interrupt vector */
    CommInItInt(COM1); /* Enable receive interrupts */
    for (;;) {
        c = CommGetChar(COM1, &err);
        if (err == COMM_NO_ERR) {
            /* Process received byte ------------------------------*/
            
            CommPutChar(COM1, ...); /* Send response */
            
        } else {
            /* Process Communication error ----------------------*/
            
        }
    }
}
11.11 Bibliography

Campbell, Joe
Sams Publishing, 1993
Indianapolis, Indiana
ISBN 0-672-30286-1

Chisvert, John P., Foster, John O.
The VISAAT Handbook
Annabooks, 1993
ISBN 0-929392-00-6

Eedelsky, Philip
“PC Interrupt-Driven Serial I/O”
From the book: MS-DOS System Programming
RAD Publications, 1990

Hallsall, Fred
Data Communications, Computer Networks and Open Systems (Third Edition)
Addison-Wesley, 1992
ISBN 0-201-56004-4

Pippenger, J.E. and Tobaben, E.J.
Linear and Interface Circuits Applications
Volume 2: Line Circuits and DISPLAY Drivers
Texas Instruments, 1985

Tannenbaum, Andrew S.
Computer Networks (Second Edition)
PTR Prentice-Hall, Inc., 1989
Listing 11.1  

```c
/* 
 * COMM_PC.C 
 * Embedded Systems building blocks 
 * Copyright 1993, John J. Lebbeus, Weston, FL 
 * All Rights Reserved 
 */

#define COMM_INIT() enable()
#define COMM_DISABLE() disable()

/* The following defines are all defined in other modules {i.e. COMM_ES, COMM_H, or COMM_IN}
 * COMM_BASE is the base address of COMM on your PC (typically 0x100)
 * COMM РФ is the number of characters buffer by the UART
 * 2 for the MS16440 
 * 16 for the MS16550 
 */

#include "include.h"
#include "COMM.h"

/* INCLUDES */
*/
Listing 11.1 (continued)  \texttt{COMMPC.C}

\begin{verbatim}
/*
 * File: \texttt{COMMPC.C}
 */

/**
 * \texttt{COMMPC.C}
 */

#define P17  0x00
#define P16  0x01
#define P15  0x02
#define P14  0x03
#define P13  0x04
#define P12  0x05
#define P11  0x06
#define P10  0x07
#define P9  0x08
#define P8  0x09
#define P7  0x0A
#define P6  0x0B
#define P5  0x0C
#define P4  0x0D
#define P3  0x0E
#define P2  0x0F
#define P1  0x10
#define P0  0x11

#define PIC_IN Mare_PORT  0x20
#define PIC_MES Mare_PORT  0x21
#define COM0_USART 0
#define COM1_USART 0
#define COM2_USART 0
#define COM3_USART 0
#define COM4_USART 0
#define COM5_USART 0
#define COM6_USART 0
#define COM7_USART 0

static pinMode(CMD111D_Port, INPUT);  
static pinMode(CMD112D_Port, OUTPUT); 
static pinMode(CMD113D_Port, OUTPUT); 
static pinMode(CMD114D_Port, OUTPUT);  

"SPI welcoming"
\end{verbatim}
Listing 11.1 (continued) COMM_PC.C

/*
   ***************************************************************************/
   /*
   * CONFUSE PORT
   */
   /*
   * Description: This function is used to configure a serial I/O port. This code is for IBM-PS/2 and
   * compatibles and assumes a National Semiconductor NS16850.
   */
   /*
   * Arguments: ch: is the COMM port channel number and can either be:
   *   COM1
   *   COM2
   *   'baud': is the desired baud rate (anything, standard rates or custom)
   *   'bits': defines the number of bits used and can be either 5, 6, 7 or 8.
   *   'parity': specifies the parity to use.
   *   COM_PARITY_EVEN
   *   COM_PARITY_ODD
   *   COM_PARITY_NONE
   *   COM_PARITY_WHITE
   *   'stop': defines the number of stop bits used and can be either 1 or 2.
   */
   /*
   * Return: COMM_OK: if the channel has been configured.
   *   COMM_BAD_CH: if you have specified an incorrect channel.
   */
   /*
   * Notes:
   * 1) Refer to the 8254410 Data sheet.
   * 2) The constant 115200 is based on a 1.8432 MHz crystal oscillator and a 14 x Clock.
   * 3) 'sgr' is the Line Control Register and is defined as:
   *   97 96 95 86 85 81 80
   *   -- -- -- --
   *   Bits (00 = 5, 01 = 6, 10 = 7 and 11 = 8)
   *   -- -- --
   *   'stop': (0 = 1 stop, 1 = 2 stop)
   *   -- --
   *   'parity': (0 = parity disabled, 1 = parity is enabled)
   *   --
   *   'sgr': when set to 1.
   *   --
   *   'sgr': when 0.
   * 4) This function enables all interrupts but only 7x interrupts.
   */
   ***************************************************************************/
Listing 11.1 (continued)  COMM_PC.C

/* Read rate device */

/* Low Control Register */

/* COM port base address */

/* Obtain base address of COM port */

switch (id) {
    case COM1: base = COM1_BASE; break;
    case COM2: base = COM2_BASE; break;
    default: return (COMM_SWITCH);
}

/* compute divisor for desired baud rate */

/* Split divisor into 16 and 43200 bytes */

/* 8 or 16 bits */

/* 128, 256, 512, or 1024 */

/* Odd parity */

/* Even parity */

/* Set divider seven bit */

/* Load divisor */

/* Set line 8088 register (Bit 8 is 0) */

/* Assert 485 and RTS and, allow interrupts */

/* Disable both Rx and Tx interrupts */

/* Push the Rx input */

"/\"
Listing 11.1 (continued)  COMM_PC.C

/*
 * Description: This function processes an interrupt from a COMM port. The function verifies whether the
 * interrupt comes from a received character, the completion of a transmitted character or
 * both.
 * Arguments: ch is the COMM port channel number and can either be:
 *                COMP
 *                COMP
 * Notes: switch statements are used for expansion.
 */

void CommHandler (INT8U ch)
{
  INTRU ch;
  INTRU irr;  /* Interrupt Identification Register (190) */
  INTRU stat;
  INTRU base;  /* COMM port base address */
  INTRU num;
  INTRU max;  /* Max. number of interrupts serviced */
Listing 11.1 (continued)  

```c
switch (ih) {
    case COM2:
        base = COM2_BASE;
        break;
    case COM3:
        base = COM3_BASE;
        break;
    default:
        base = COM3_PROP;
        break;
}

max = COM3_PROP;

if ((DATA[base + COM3_PROP + COM3_PROP] & 0x87) /* Get contents of ISR */
    process ALL interrupts /*
    [Code]
    
    case 0: /* See if we have a Tx interrupt */
        if (max == COM3_PROP) /* If we have a Tx interrupt */
            outputBase + COM3_PROP, stat); /* Get data to output */
        break;

    case 1: /* Get if we have a Rx interrupt */
        if (max == COM3_PROP) /* If we have a Rx interrupt */
            outputBase + COM3_PROP, stat); /* Get data to output */
        break;

    case 2: /* If we have a Rx interrupt */
        if (max == COM3_PROP) /* If we have a Rx interrupt */
            outputBase + COM3_PROP, stat); /* Get data to output */
        break;

    case 3: /* Get contents of ISR */
        if (max == COM3_PROP) /* Get contents of ISR */
            outputBase + COM3_PROP, stat); /* Get data to output */
        break;

    default:
        outputBase + COM3_PROP, (stat); /* Reset int=744 controller */
        break;
}
```

Chapter 11: Asynchronous Serial Communications — 401
Listing 11.1 (continued)  **COMM_FC.C**

/*
   * Description: This function restores the old interrupt vector for the desired communication channel.
   * Arguments:  
   *   ch: is the COM port channel number and can either be:
   *       0: COM1
   *       1: COM2
   * Notes:  
   *       This function assumes that the board is running in Real mode.

   */

void CommInterrupt(int ch)
{
    DECB *pvec;

    switch (ch) {
    case COM1:
        pvec = (DECB *)0x400d0000, 0x00 << 3); /* Pointer to proper INT location */
        pvec-> = COM1INT(0x0000);
        pvec-> = COM1INT(0x0000);
        break;
    case COM2:
        pvec = (DECB *)0x400d0000, 0x00 << 3); /* Pointer to proper INT location */
        pvec-> = COM2INT(0x0000);
        pvec-> = COM2INT(0x0000);
        break;
    }

    /*IN32*/
Listing 11.1 (continued)  COMM_PC.C

/*/  **********************************************************************************************/
/*  */
/* DESCRIPTION : This function is called to flush any input characters still in the receiver. This */
/* function is useful when you replace the MS2450 with the more powerful MS2455. */
/* Arguments : sbr : is the COMM port channel number and can either be */
/* COMM1 */
/* COMM2 */
/* **********************************************************************************************/
*/
void COMMFlush (int sbr)
{
    int i;
    int bmore;

    switch (sbr)
    {
    case COMM1:
        bmore = COMM_SIZE;
        break;
    case COMM2:
        bmore = COMM_SIZE;
        break;
    }
    CLR  = COMM_CHAR_RX;
    do interrupt (critical);  
    while (bmore > 0)  
    {  
        bmore = bmore - 1;
    }
    ds interrupt (critical);

    /*SPACE*/
Listing 11.1 (continued) \texttt{COMM\_PC\_C}

\begin{verbatim}
/*
 * Description: This function disables the Rx interrupt.
 * Arguments: ch is the COM port channel number and can either be:
 * COM1
 * COM3
 */

void CommWrite(int ch)
{
    DWORD stat;

    switch (ch) {
    case COM1:
        oh_COM_putchar(); /* Disable Rx interrupts */
        outp(COMMBASE + COMM_UART_ID, &RXI); /* Both Tx & Rx interrupts are disabled */
        if (stat = bXOR) { /* Yea, disable IRQ on the PIC */
            outp(FIC_MSR_REG_PORT, (DWORD)inp(FIC_MSR_REG_PORT) | BT3);
            OE_INT2_CRITICAL();
        }
        break;
    case COM3:
        oh_COM_putchar(); /* Disable Rx interrupts */
        outp(COMMBASE + COMM_UART_ID, &RXI); /* Both Tx & Rx interrupts are disabled */
        if (stat = bXOR) { /* Yea, disable IRQ on the PIC */
            outp(FIC_MSR_REG_PORT, (DWORD)inp(FIC_MSR_REG_PORT) | BT3);
            OE_INT2_CRITICAL();
        }
        break;
    }
}
/*lP1*/
\end{verbatim}
void Comm Phần 3(ch)
{
  void
  short st;

  switch (ch)
  {
  case 0:
    case 1:
      case 2:
      case 3:
    case 4:
      case 5:
      case 6:
      case 7:
      case 8:
      case 9:
        break;
        break;
        break;
        break;
        break;
        break;
        break;
        break;
        break;
        break;
        break;
        break;
  
  default:
    strcpy(Comm Port[ch], RETR);
    break;
  
  }
Listing 11.1 (continued)  COMM_FC.C

void COMMIntVec (2MBio db)
{
  IN160 =spext;
  IN161 =offs;
  IN162 =pvec;

  switch (db) {
    case COMM:
      pvec = (32Reg) *HPI_start +5000, 0x6C << 2; /* Point to proper IFP location */
      Case22590(ISIRVECTOR(1));
      Case22594(ISIRVECTOR(2)); /* Save current vector */
      Case22598(ISIRVECTOR(3));
      pvec = (16Reg) *HPI_start +1500;
      pvec = (16Reg) *HPI_start +1250;
      Case22604(ISIR_ALL); /* Set new vector */
      Case22608(ISIR_ALL);
      Case22612(ISIR_ALL); /* Save vector */
      Case22616(ISIR_ALL); break;
    case COMM2:
      pvec = (32Reg) *HPI_start +5000, 0x6C << 2; /* Point to proper IFP location */
      Case22640(ISIRVECTOR(1));
      Case22644(ISIRVECTOR(2)); /* Save current vector */
      Case22648(ISIRVECTOR(3));
      pvec = (16Reg) *HPI_start +1500;
      pvec = (16Reg) *HPI_start +1250;
      Case22654(ISIR_ALL); /* Set new vector */
      Case22658(ISIR_ALL); break;
  }

  /*END*/
}
Listing 11.1 (continued)  

```c
/*
* Description: This function disables the character transmission.
* Arguments: 'ch' is the COM port channel number and can either be:
*      COMM6
*      COMM2
*
*/

void Comm0ConfDis (DMMU ch)
{
    DMMU stat;
    DMMU cmd;

    switch (ch) {
    case COMM6:
        OS_ENTER_CRITICAL();
        /* Disable Tx interrupts */
        stat = (DMMU)(ip)(COM6BASE + COM6_INTR_HIR) & ~BIT7;
        output(COM6BASE + COM6_INTR_LIR, stat);
        if (stat == 0) {
            cmd = (DMMU)(ip)(PIC6_IRQ_CNTL_PORT) & BIT7;
            output(PIC6_IRQ_CNTL_PORT, cmd);
            /* Yes, disable IRQ4 on the PC */
            OS_EXIT_CRITICAL();
            break;
        }
        break;
    case COMM2:
        OS_ENTER_CRITICAL();
        /* Disable Tx interrupts */
        stat = (DMMU)(ip)(COM2BASE + COM2_INTR_HIR) & ~BIT7;
        output(COM2BASE + COM2_INTR_LIR, stat);
        if (stat == 0) {
            cmd = (DMMU)(ip)(PIC2_IRQ_CNTL_PORT) & BIT7;
            output(PIC2_IRQ_CNTL_PORT, cmd);
            /* Yes, disable IRQ3 on the PC */
            OS_EXIT_CRITICAL();
            break;
        }
    } /*UNZIP*/
```
Listing 11.1 (continued)  COMM_PC.C

/*
** DESCRIPTION: This function enables transmission of characters. Transmission of characters is
** interrupt driven. If you are using a multi-drop driver, the code must enable the driver
** for transmission.
**
** ** Arguments:
** 
** ch is the COM port channel number and can either be:
** 
** COM1/COM2
**
*/

void CommTrans(int ch)
{
    int state;
    int reg;

    switch (ch) {
        
        case COM1:
            // Enable COM1
          /* Enable Tx interrupts*/
            break;
        case COM2:
            // Enable COM2
          /* Enable Tx interrupts */
            break;
        default:
            // Default
            break;
    }
}

Listing 11.2  COMM_PC.H

/* ................................................................. */
/* Embedded Systems Building Blocks                        */
/* Complete and Ready-to-Use Modules in C                  */
/* Asynchronous Serial Communication                      */
/* EZO-PC Serial UART Low Level Drivers                   */
/* (c) Copyright 1999, Joes P. Lousbro, Naples, FL        */
/* All Rights Reserved                                     */
/* Filename : COMM_PC.H                                     */
/* Programmer : Joes P. Lousbro                           */
/* .............................................................................. */
/* */
/* .............................................................................. */

/* */

/* CONFIGURATION CONSTANTS */

/* */

#undef CPU_H

#define COMM_Base  0x4100  /**< base address of PC = COM1 */
#define COMM_Base  0x4120  /**< base address of PC = COM2 */
#define COMM_Buffer 2  /**< NS1455 has 2 byte buffer */

#endif

/* */

/* */

/* */

void CommInit(void);  /**< */
void CommSend(void);  /**< */
void CommReceive(void);  /**< */
void CommSetTransmitMode(void);  /**< */
void CommStopTransmit(void);  /**< */
void CommStopReceive(void);  /**< */
void CommSetDatabits(void);  /**< */
void CommSetParity(void);  /**< */
void CommSetStopbits(void);  /**< */
void CommSetBaudRate(void);  /**< */
void CommSetOutRecMode(void);  /**< */
void CommSetOutRecMode(void);  /**< */
void CommSetOutRecMode(void);  /**< */
void CommSetOutRecMode(void);  /**< */
**Listing 11.3 COMM_PCA.ASM**

```asm
; Complete and Ready-to-Use Modules in C
; Asynchronous Serial Communications
; IBM-PC Serial I/O low level Driver
;
(c) Copyright 1999, Jean J. Labrosse Weston, FL
; All Rights Reserved

; Filename : COMM_PCA.ASM
; Programmed : Jean J. Labrosse
; Notes : if you are not using sIOsett: you will need to DELETE the increments of okIOsett and
; the calls to DEFIOsett().

PUBLIC _CommISR
PUBLIC _CommISR заруб
EXTERN _CommISR заруб зар
EXTERN _CommISR заруб зар зар
EXTERN _CommISR зар

MODEL LARGE
.CODE
.OFF
.localized

;*/SOURCE*/
```
Listing 11.3 (continued)  COMM_PCA.ASM

 /******************************************************************************
 |
 | COMM_ISR PROC far
 | PUSHA
 | PUSH DS
 | PUSH ES
 | MOV AX, DEBUG
 | MOV DS, AX
 | ; NOTE: Comment out the next line (i.e. MOV [BX][DI]Debug) if you don't use DEBUG-12.
 | MOV [BX][DI]Debug ; Notify uC/OS-II of ISR
 | POP DS ; Indicate ISR
 | CALL FAR PTR COMMISRHandler ; Process ISR interrupt
 | ADD SP,2 ;
 | ; NOTE: Comment out the next line (i.e. CALL DebugExit if you don't use DEBUG-12.
 | CALL FAR PTR DebugExit ; Notify OS of end of ISR
 | POP DS ; Restore interrupted task's context
 | POP ES
 | POPA
 | SHOWISR DBP
 | /*ISR*/
 | */
Listing 11.3 (continued) COMM_FCA.ASM

;---------------------------------------------------------
; MAURICE COMM ISR
;---------------------------------------------------------
CommISR PROC FAR

; Save interrupted task's context
PUSH AX
PUSH DS
PUSH SI
PUSH DI
MOV AX, DS:0
MOV DS, AX

; Note: Comment out line 14 (i.e., INC_PTR_DistInterrupt) if you don't use uCODE-11.
INC PTR_PTR_DistInterrupt
; Notify uCODE-11 of ISR

PUSH 2
; Indicate COMM
CALL PTR_PTR_CommInterrupt
ADD SP, 2

; Note: Comment out line 16 (i.e., CALL_PTR_DistExit) if you don't use uCODE-11.
CALL FAR_PTR_PTR_DistExit
; Notify OS of end of ISR

POP DI
; Restore interrupted task's context
POPF

; Return to interrupted task
End CommISR

END
Listing 11.4 COMMEND.C

/* **************************************************************************
 * Embedded System Building Blocks
 * Complete and Ready-to-Use Modules in C
 * Asynchronous Serial Communications
 * Buffered Serial I/O
 * (foreground/background Systems)
 * Copyright 1999, John J. Labrosse, Isoton, FL.
 * All Rights Reserved
 *
 * Filename : COMMEND.C
 * Programmer : John J. Labrosse
 *
 * Notes :
 * the functions (actually macros) DEFgnore_CRITICAL() and DEFgnore_CRITIION() are used to
 * disable and enable interrupts, respectively. If using the Borland C compiler V5.1,
 * all you need to do is to define these macros as follows:
 * #define DEFgnore_CRITICAL() disable()
 * #define DEFgnore_CRITIION() enable();
 */

/* **************************************************************************
 * INCLUDES
 *
 *
 */

#include "includes.h"

"SPEND"
Listing 11.4 (continued) COMM.GND.C

/*
  **********************************************************************************************
  *                                      CONTENTS                                               *
  *                                                                                             *
  **********************************************************************************************
  */

  typedef struct {
    int32U RingBufLen;        /* number of characters in the Rx ring buffer */
    int32U *RingBufInPtr;    /* pointer to where next character will be inserted */
    int32U *RingBufOutPtr;   /* pointer from where next character will be extracted */
    int32U RingBuf[COMM_RX_BUF_SIZE]; /* Ring buffer character storage (Rx) */
    int32U RingBufLenRx;     /* number of characters in the Rx ring buffer */
    int32U *RingBufInPtrRx;  /* pointer to where next character will be inserted */
    int32U *RingBufOutPtrRx; /* pointer from where next character will be extracted */
    int32U RingBuf[COMM_TX_BUF_SIZE]; /* Ring buffer character storage (Tx) */
  } COMM_RING_BUF;

  /*****************************************************************************/
  *
  /*******************************************************************************/
  
  /* COMM_RING_BUF CommRing;
  COMM_RING_BUF CommRing;
  */

  void CommRing_Init(void)
  { /* ... */
    COMM_RING_BUF CommRing;
    CommRing.Init();
  }

  /*****************************************************************************/
  /*
  ******************************************************************************/
Listing 11.4 (continued) COMMAND.C

/*******************************************************************************/
/* COMMAND.C - COMMON COMMAND RFU CHARACTERS FOR RFU MASTER */
/*******************************************************************************/

/* Description: This function is called by your application to obtain a character from the communications
   channel. */

/* Arguments: */
/* 'ch' is the COM port channel number and can either be: */
/* COM1 */
/* COM2 */
/* 'err' is a pointer to where an error code will be placed. */
/* 'err' is set to COM0_NULL if a character is available. */
/* 'err' is set to COM0_NULL if the Rx buffer is empty. */
/* 'err' is set to COM0_NULL if you have specified an invalid channel */

/* Returns: */
/* The character in the buffer can get if the buffer is empty */

/*******************************************************************************/

#define COM0_NULL {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}

int ComGetChar (char ch, int *err)
{
    int
    COM0_NULL "*plbf;

    Which ch
    {
        case COM1:
            plbf = &Com1bf;
            break;
        case COM2:
            plbf = &Com2bf;
            break;
        default:
            *err = COM0_NULL;
            return NULL;
    }

    if (*plbf->wBuffer[0] > 5) /* See if buffer is empty */
    {
        // No decrement character count
        *plbf->wBuffer[0] = *plbf->wBuffer[0] - 1;
        if (*plbf->wBuffer[0] > *plbf->wBuffer[1]) /* Read next character */
            *plbf->wBuffer[1] = *plbf->wBuffer[0];
    }

    *err = *plbf->wBuffer;
    return (*err);
}

else
    {
        *err = COM0_NULL;
        return NULL;
    }

    /* returns empty, return NULL */
}

/*SNET/
Listing 11.4 (continued) COMMIBND.C

/*
 * COMMIBND.C
 * COMMIBND.C
 /* Description: This function is called by the Tn ISR to extract the next character from the Tn buffer.
 * The function returns FALSE if the buffer is empty; the character is extracted from the buffer. This is done to signal the Tn ISR to disable interrupts because this is the last character to read.
 * Arguments: ch is the COMM port channel number and can either be
 * COMM0 or COMM1.
 */

#define COMMIBND(ch) COMMIBND0(ch, BUF0) /* err */

 COMMIBND(ch) { 
    void

    switch(ch) { /* Obtain pointer to communications channel */
    case COMM0:
       phbuf = &COM0Buff;
       break;
    case COMM1:
       phbuf = &COM1Buff;
       break;
    default:
       /*err = COMM furnished
       return(SERR);*/

    if (phbuf->bufLength < 0) { /* See if buffer is empty */
       bufLength = phbuf->bufLength;
       /* Get character from buffer */
       if ((phbuf->bufLength > 0) && (phbuf->bufLength < phbuf->bufLength)) {
          /* Map 000 pointer
          phbuf->bufLength = phbuf->bufLength[0];
          error = COMMBUF_ERR;
          return(SERR);*/
       }
    } else { /* Characters are still available */
       error = COMM_BUF_INTPV;
       return(SERR);
    }
    /* buffer is empty */
    
    } /*Main*/
*/

} /*COMMIBND*/
Listing 11.4 (continued)  COMMAND.C

/*
 ******************************************
 * INITIALIZE COMMUNICATIONS MODULE
 */

/*
 Description: This function is called by your application to initialize the
 communications module. You must call this function before calling any
 other functions.
 */

void CommInit (void)
{
  // COMMUNICATIONS Module
  if (COMMinit) {
    phl = W1001buf;
    phl->RingBuffer[0] = {phl->RingBuffer[0]};
    phl->RingBuffer[1] = {phl->RingBuffer[1]};
    phl->RingBuffer[2] = {phl->RingBuffer[2]};
    phl->RingBuffer[3] = {phl->RingBuffer[3]};
    phl->RingBuffer[4] = {phl->RingBuffer[4]};
    phl->RingBuffer[5] = {phl->RingBuffer[5]};
    phl->RingBuffer[6] = {phl->RingBuffer[6]};
    phl->RingBuffer[7] = {phl->RingBuffer[7]};
    //*/
  }

  /* Initialize the ring buffer for COM0 */
  phl = W1002buf;
  phl->RingBuffer[0] = {phl->RingBuffer[0]};
  phl->RingBuffer[1] = {phl->RingBuffer[1]};
  phl->RingBuffer[2] = {phl->RingBuffer[2]};
  phl->RingBuffer[3] = {phl->RingBuffer[3]};
  phl->RingBuffer[4] = {phl->RingBuffer[4]};
  phl->RingBuffer[5] = {phl->RingBuffer[5]};
  phl->RingBuffer[6] = {phl->RingBuffer[6]};
  phl->RingBuffer[7] = {phl->RingBuffer[7]};
}

/*END*/
/*
 * Description: This function is called by your application to see if any character is available from the
 * communications channel. If at least one character is available, the function returns
 * FALSE otherwise, the function returns TRUE.
 * Arguments: ch  is the COM port channel number and can either be:
 * COM1
 * COM2
 * Returns: TRUE if the buffer is not empty,
 * FALSE if the buffer is not empty and you have specified no (BUFFERoodles)
 *>*******************************************************************************/

BOOLOAL CommIsEmpty (INT8U ch)
{
    BOOL bool empty;
    COM_SIOCOMM *pcom;

    match(ch) { /* Obtain pointer to communications channel */
        case COM1:
        pcom = &comch1;
        break;
        case COM2:
        pcom = &comch2;
        break;
        default:
        return FALSE;
    }
    OR_ELSE dangerously();
    if (pcom->COUNT > 0) { /* See if buffer is empty */
        empty = FALSE;
    } else { /* Buffer is not empty */
        empty = TRUE;
    }
    OR_ELSE dangerously();
    return (empty);
}

/*Send*/
Listing 11.4 (continued)  COMMAND.C

/*
 * SEE IF THE CHARACTER QUEUE IS FULL.
 */

/*
 * Description: This function is called by your application to see if any more characters can be placed in the character queue. In other words, this function checks to see if the character queue is full.
 * If the character queue is full, the function returns TRUE otherwise it returns FALSE.
 * Arguments: "cmsg" is the message queue number and "cmd" the command.
 * Returns: TRUE if the queue is full.
 *          FALSE if the queue is NOT full or you have specified no input port channel
 */

#define CommandInlU (cmsg, cmd) do
{ flush (cmd); 
  COMMAND *pmsg; 

  switch (cmd) { 
  case COMIN1U: 
  pmsg = comin1uf(); 
  break; 
  case COMIN2U: 
  pmsg = comin2uf(); 
  break; 
  default:
    return (FALSE); 
  }
  if (pmsg->isnull || pmsg->cmsg != cmsg) { 
    /* see if buffer is full */
    return (TRUE);
  } else { 
    /* buffer is NOT full */
    return (FALSE); 
  }
} /* IN1U*/

/""/*IN2U*/

/*
Listing 11.4 (continued)  COMMSEND.C

/*
  *******************************************************************************/

/*
   Description : This function is called by your application to send a character on the communications
   channel. The character to send is first UNicode into the tx buffer and will be sent by
   the IOM. If this is the first character placed into the buffer, the tx IOM will be
   enabled. If the tx buffer is full, the character will get not be sent (i.e., it will be lost).

   Arguments : ch is the COM port channel number and can either be
   0
   or
   1
   ch

   c is the character to send.

   Returns : COMM_NO_ERR if the function was successful (the buffer was not full)
   or
   COMM_TX_FULL if the buffer was full
   or
   COMM_ERR if you have specified an incorrect channel
  *******************************************************************************/

int COMMSEND (IOM ch, char c)
{
    switch (ch) {
    case 0:
        if (CommBuf[0] == 0)
            return (COMM_WAITING);
        if (CommBuf[1] == 0)
            return (COMM_WAITING);
        break;

    case 1:
        if (CommBuf[2] == 0)
            return (COMM_WAITING);
        if (CommBuf[3] == 0)
            return (COMM_WAITING);
        break;

    default:
        return (COMM_ERR);
    }

    if (CommBuf[0] + 1 > COMM_BUF_SIZE) /* See if buffer is full */
        return (COMM_TX_FULL);
    if (CommBuf[1] + 1 > COMM_BUF_SIZE) /* See if buffer is full */
        return (COMM_TX_FULL);
    if (CommBuf[2] + 1 > COMM_BUF_SIZE) /* See if buffer is full */
        return (COMM_TX_FULL);
    if (CommBuf[3] + 1 > COMM_BUF_SIZE) /* See if buffer is full */
        return (COMM_TX_FULL);

   _COMM_SEND_INTR(1);
    return (COMM_NO_ERR);
}

/*UCP*/
Listing 11.4 (continued) COMMAND.C

/**
 * COMMAND.C
 *
 * Description: This function is called by the ISR to insert a character into
 * the receive ring buffer.
 * Arguments: ch - the character to insert into the ring buffer.
 *            ch is the COM port channel number and can either be:
 *            COM0      
 *            COM1      
 *            'c' - the character to insert into the ring buffer. If the
 *            buffer is full, the character will not be inserted, it will be
 *            lost.
 */

void CommandChar (unsigned ch, int ch) 
{
    switch (ch) 
    {
    case COM0:
        phuf = &COM0buf;
        break;
    case COM1:
        phuf = &COM1buf;
        break;
    default:
        return;
    }
    if (phuf->length > phuf->size)  // See if buffer is full
        phuf->length = phuf->size;  // No increase, decrease count
    phuf->length++;  // No change, just buffer
    if (phuf->length + 1 > phuf->size)  // * Wrap IN pointer
        phuf->length = phuf->size;  
}
Listing 11.5 COMM.BND.H

/*
 * Embedded Systems Building Blocks
 * Complete and Ready-to-Use Modules in C
 * Asynchronous Serial Communications
 * Buffered Serial I/O
 * (URGently Background System)
 * 
 * (C) Copyright 1995, Juan J. Lebarba, Weston, FL
 * All Rights Reserved
 * 
 * Filename : COMM.BND.H
 * Programmer : Juan J. Lebarba
 * 
 * CONFIDENTIAL CONTENTS
 */

/*

#define COM_BUFF_SIZE 128 /* Number of characters in Rx ring buffer */
#define COM_TX_BUFF_SIZE 128 /* Number of characters in Tx ring buffer */

 gratuitement

#define MLL 0001
#endif

#define COMM 1
#define COMM2 2
#define COMM_RX_OK 0 /* Function call was successful */
#define COMM_RX_OK 1 /* Successfully transmitted packet */
#define COMM_RX_OK 2 /* Tx buffer is empty, no character available */
#define COMM_RX_OK 3 /* Tx buffer is full, could not deposit character */
#define COMM_RX_OK 4 /* If Rx buffer is empty. */

#endif COMM_DEBUG

#define COMM_DEBUG #define COMM_Err #define COMM_EXIT /*COMM.EXIT return */
#ifndef /*/
Listing 11.5 (continued) COMMGRID.H

/****************************************************************************
** FUNCTION PROTOTYPES
****************************************************************************/

.NumericUpDown CommGetCount(uintch, DWORD *arg);
.NumericUpDown CommSpaceCount(DWORD ch, DWORD *arg);
void CommInit(void);
BOOL88 CommInitSync(DWORD ch);
BOOL88 CommInitSync(DWORD ch);
DWORD CommFtData(DWORD ch, DWORD 0);
void CommNotifyChange(DWORD ch, DWORD 0);
Listing 11.6  COMMRTO.S.C

/*
   ****************************************************************************
   *  Embedded Systems Building Blocks
   *  Complete and Read-to-Use Modules in C
   *  Asynchronous Serial Communications
   *  Buffered Serial I/O
   *  (RTS)
   *
   *  (c) Copyright 1999, Jean J. Labrosse, Weston, FL
   *  All Rights Reserved
   *
   * File:       COMMRTO.S.C
   * Programmed: Jean J. Labrosse
   ****************************************************************************
   */

#include "includes.h"

/*
   ****************************************************************************
   * INCLUDES
   *
   * GLOBAL VARS
   *
   * */

typedef struct {
  void * RingBufH: /* Number of characters in the Rx ring buffer */
  char * RingBufPtr: /* Pointer to be semaphore */
  void * RingBufRd: /* Pointer to where next character will be inserted */
  void * RingBufWr: /* Pointer from where next character will be extracted */
  void * RingBufEnd: /* Ring buffer character storage (64) */
  void * RingBufSz: /* Number of characters in the Rx ring buffer */
  char * RingBufRd: /* Pointer to Rx semaphore */
  void * RingBufWr: /* Pointer to where next character will be inserted */
  void * RingBufEnd: /* Pointer from where next character will be extracted */
  void * RingBufSz: /* Ring buffer character storage (64) */
} RFJ_NULL_BUF;

GLOBAL VARIABLES

COM_RINGBUF_Coord:
COM_RINGBUF_Coord:

/*DONE*/
Listing 11.8 (continued)  CONRTOS.C

/*

**************************************************************************

* Description: This function is called by your application to obtain a character
* from the communications channel. The function will wait for a character to be
* received on the serial channel or until the function times out.
*
* Arguments: ch: is the COM port channel number and can either be:
* COM1
* COM2
* wr: is the amount of time (in clock ticks) that the calling function is willing to
* wait for a character to arrive. If you specify a timeout of 0, the function will
* wait forever for a character to arrive.
* *err is a pointer to where an error code will be placed. If an error occurs
* *err is set to COM.ERR_TIMEOUT if a timeout occurred
* *err is set to COM.ERR_BLOCK if you specify a invalid channel number
*
* Returns: The character in the buffer for wr. If a timeout occurred

**************************************************************************/

POKE COMGetChar (WORD ch, INCH wr, INCH *err) returns INCH

DBUF c;
err;
COM_SR_READ ’DBUF’;

ENTRY ch()
call COM_SR;
   DBUF +4COMBuf;
   break;
next:
   if COM_SR
   DBUF +8COMBuf;
   break;
   default:
   *err = COM.ERR_BLOCK
   return 0;
   {}
   if COM_SR
   *err = COM.ERR_TIMEOUT
   return -1;
   {}
   if COM_SR
   else if COM_SR
   else if COM_SR
   {}
   else
   {}
   {}
   {}
   {}
   {}
   return 0;
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   {}
   }

'
Listing 11.6 (continued) COMRTOS.C

```c
/*
** Remaining functions
**
** GETキャラクター FROM SIGNED BUFFER
**
** Description: This function is called by the Tx ISR to extract the next character from the Tx buffer.
** The function returns TRUE if the character is extracted from the buffer. This is done to signal the Tx ISR to disable interrupt because this is the
** last character to send.
** Arguments: 
** COM is the COM port channel number and can either be:
** COM0
** COM1
** 'buf' is a pointer to where an INTERR code will be deposited.
** 'err' is set to COMN_CORRUPT if at least one character was available
** from the buffer.
** 'err' is set to COMN_XFER if the Tx buffer is empty.
** 'err' is set to COMN_TXCHG if you have specified an incorrect channel
**
** Returns: The next character in the Tx buffer or NULL if the buffer is empty.
**
*/

#define COMN_CORRUPT 0x01
#define COMN_XFER 0x02
#define COMN_TXCHG 0x03

switch (ch) {
    case COM0:
        if (buf->s瞻gflagTxFtcr == 0) { /* See if buffer is empty */
            buf->s瞻gflagTxFtcr = buf->s瞻gflagRxFtcr;
            ch = buf->s瞻gflagTxFtcr; /* Get character from buffer */
        } else if (buf->s瞻gflagTxFtcr - buf->s瞻gflagRxFtcr < 2) { /* How many characters */
            buf->s瞻gflagTxFtcr = buf->s瞻gflagRxFtcr;
            ch = buf->s瞻gflagTxFtcr; /* Indicate that character will be sent */
        } else { /* Characters are still available */
            return (com); /* Buffe is empty */
        }
    case COM1:
        if (buf->s瞻gflagTxFtcr == 0) { /* See if buffer is empty */
            buf->s瞻gflagTxFtcr = buf->s瞻gflagRxFtcr;
            ch = buf->s瞻gflagTxFtcr; /* Get character from buffer */
        } else if (buf->s瞻gflagTxFtcr - buf->s瞻gflagRxFtcr < 2) { /* How many characters */
            buf->s瞻gflagTxFtcr = buf->s瞻gflagRxFtcr;
            ch = buf->s瞻gflagTxFtcr; /* Indicate that character will be sent */
        } else { /* Characters are still available */
            return (com); /* Buffe is empty */
        }
    }
```

Listing 11.6 (continued) COMPTOS.C

```c
/*
 * Description: This function is called by your application to initialize the communications module. You
 * must call this function before calling any other functions.
 * Arguments: none
 */

void COMINIT (void)
{
    phuf = aCommBUF; /* Initialize the ring buffer for COM1 */
    phuf ->RingBufCtx = 0;
    phuf ->RingBufInit = phuf ->RingBufInit1C;
    phuf ->RingBufMode = COM1RxMode;
    phuf ->RingBufContext = 0;
    phuf ->RingBufTimeout = 0;
    phuf ->RingBufDoneISR = phuf ->RingBufDoneISR1;
    phuf ->RingBufDoneISR1 = 0;
    /* Create a new COM1_RX_ISR */
    phuf->RingBufDoneISR1 = COM1_RX_ISR;
    phuf = aCommBUF; /* Initialize the ring buffer for COM2 */
    phuf ->RingBufCtx = 0;
    phuf ->RingBufInit = phuf ->RingBufInit2C;
    phuf ->RingBufMode = COM2RxMode;
    phuf ->RingBufContext = 0;
    phuf ->RingBufTimeout = 0;
    phuf ->RingBufDoneISR = phuf ->RingBufDoneISR2;
    phuf ->RingBufDoneISR2 = 0;
    /* Create a new COM2_RX_ISR */
    phuf->RingBufDoneISR2 = COM2_RX_ISR;
}
*/
```
Listing 11.6 (continued)  COMMRTOS.C

/****
 * SEE IF CHARACTER BUFFER IS EMPTY
 * *
 * Description : This function is called by your application to see if any character is available from the
 * communications channel. If at least one character is available, the function returns
 * POSH otherwise, the function returns TRUE.
 * Arguments : ch  the COMM port channel number and one either be:
 *            COM1
 *            COM2
 * Returns : TRUE  if the buffer is empty.
 *           FALSE  if the buffer is NOT empty or you have specified an incorrect channel.
 ****/

#define CommEmpty (COMx) do
{
    switch (inh) {
    case COM1:
        pbuf = &Com1Buf;
        break;
    case COM2:
        pbuf = &Com2Buf;
        break;
    default:
        return (TRUE);
    }
    if (pbuf->lengthOfBuf > 0) {
        if (pbuf->lengthOfBuf > 0) {
            empty = FALSE;
        } else {
            empty = TRUE;
        }
    } else {
        empty = TRUE;
    } return (empty);

/*$\$END$*/
/**
 * USE If the COM PORT BUFFER IS FULL.
 * 
 * Description: This function is called by your application to see if any more characters can be placed
 * in the TX buffer. In other words, this function checks to see if the TX buffer is full.
 * 
 * Arguments: __buf__ is the COM port channel number _&COMx_.
 * __BSTOP__
 * __CMODE__
 * 
 * Returns: __TRUE__ if the buffer IS full.
 * __FALSE__ if the buffer IS NOT full OR if you have specified no transmit channel.
 * 
 */

void COM/full(PGMU6-ch)
{
    EXCLAM
    CALL
    __CMODE__ *buf; /* Obtain pointer to communications channel */
    switch (__buf) {
    case __CMODE__:
        *buf = __CMODE__;
        break;
    case __CMODE__:
        *buf = __CMODE__;
        break;
    default:
        *buf = __CMODE__;
        break;
    }
    if ((__buf) && __CMODE__ *buf + _COM_TX_BUF_SIZE) {
        return nil; /* See if buffer is full */
    } else {
        return nil; /* Buffer is full */
    }
    return nil; /* Buffer is full */
    /* STOP*/
Listing 11.6 (continued) COMMRTOS.C

/*
* "COMMRTOS.C"
*
* Description: This function is called by your application to send a character on the communications channel.
* The function will wait for the buffer to empty out if the buffer is full.
* The function returns to your application if the buffer does not empty within the specified timeout.
* A timeout value of 0 means that the calling function will wait forever for the buffer to empty out.
* The character to send is first inserted into the TX buffer and will be sent by the TX ISR.
* If this is the first character placed into the buffer, the TX ISR will be enabled.
* Arguments:
* 'c' is the character to send.
* 'ch' is the character port number and can either be COM0 or COM1.
* 'to' is the timeout (in clock ticks) to wait in case the buffer is full. If you specify a timeout of 0, the function will wait forever for the buffer to empty.
* Returns:
* '0' if the character was placed in the TX buffer
* '1' if the buffer did not empty within the specified timeout period
* '10' if you specify an invalid channel number
*
*/

COMM SendChar (char c, char ch, int to)
{
  unsigned int chnr = ch;

  switch (ch) { /* Obtain pointer to communications channel */
    case COM0:
      chnr = COM0_D0;
      break;
    case COM1:
      chnr = COM1_D0;
      break;
    default:
      return COMM_ERR_DH;
  }

  c = (unsigned char) c;

  if (c == 0x0D) /* Inject CR char */
    return COMM_ERR_DH;

  if (c == 0x0A) /* Inject LF char */
    return COMM_ERR_DH;

  if (c == 0x08) /* Clear Char */
    return COMM_ERR_DH;

  if (c == 0x09) /* Inh Char */
    return COMM_ERR_DH;

  if (c == 0x10) /* ScrollChar */
    return COMM_ERR_DH;

  if (c == 0x11) /* SwitchChar */
    return COMM_ERR_DH;

  if (c == 0x12) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x13) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x14) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x15) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x16) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x17) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x18) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x19) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1A) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1B) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1C) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1D) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1E) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x1F) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x20) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x21) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x22) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x23) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x24) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x25) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x26) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x27) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x28) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x29) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2A) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2B) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2C) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2D) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2E) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x2F) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x30) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x31) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x32) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x33) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x34) /* SelectChar */
    return COMM_ERR_DH;

  if (c == 0x35) /* SelectCh
Listing 11.6 (continued) COMRTOS.C

/*
 * INSERT CHARACTER INTO RING BUFFER
 *
 * DESCRIPTION: This function is called by the INT 19H to insert a character into the receive ring buffer.
 * Arguments: 'ch' is the COM port character number and can either be:
 * COM0
 * COM1
 * 'c' is the character to insert into the ring buffer. If the buffer is full, the
 * character will not be inserted, it will be lost.
 */

void ComPutChar (INT8U ah, INT8U ch, INT8U c);
{
    switch (ah) {
        case COM0:
            if (ah->RingBufCntr + COM_BUF_SIZE <= ah->RingBufSize) { /* See if buffer is full */
                ah->RingBuf[ah->RingBufCntr] = c; /* Put character into buffer */
                ah->RingBufCntr++; /* No increment character count */
            }
            break;
        case COM1:
            if (ah->RingBuf[ah->RingBufCntr] == 0) ah->RingBuf[ah->RingBufCntr] = c; /* Put character into buffer */
            if (ah->RingBuf[ah->RingBufCntr] == 0) ah->RingBufCntr = 0; /* wrap IN pointer */
            break;
        default:
            return;
    }
    ComInterrupt(ah->RingBufDone); /* Indicate that character was received */
}
/**
 * Embedded Systems Building Blocks
 * Complete and Ready-to-use libraries in C
 *
 * Synchronous Serial Communications
 * Buffered Serial I/O
 *
 * (c) copyright 1989, Jean J. Lebrasse, Weston, FL
 * All Rights reserved
 *
 * Filename : COMMRTOS.H
 * Programmer : Jean J. Lebrasse
 */

/*
 * CONFIGURATION CONTENTS
 */

#define CRH 64 /* Number of characters in Ring buffer */
#define CRH 64 /* Number of characters in Ring buffer */

#endif

#define HEL 250
#endif

#define COMS 1 /* COMS CODES */
#define COMS 3 /* COMS CODES */

#define COMS 0 /* Debug call was successful */
#define COMS 1 /* Debug communications port channel */
#define COMS 2 /* Rx buffer is empty, no character available */
#define COMS 3 /* Tx buffer is full, could not deposit character */
#define COMS 4 /* If the Rx buffer is empty */
#define COMS 5 /* If a timeout occurred while waiting for a character */
#define COMS 6 /* If a timeout occurred while waiting to send a char */

#define COMS 0 /* Defines for setting parity */
#define COMS 1 /* COMS CONTENTS */
#define COMS 2 /* COMS CONTENTS */
Listing 11.7 (continued) COMMRTOS.H

#define COMM_TIMEOUT 
#define COMM_Read 
#define COMM_Write 

/* */

COMM_Init();

COMM_Close();

COMM_Open();

COMM_Transmit();

COMM_Receive();

COMM_End();

COMM_Open();

COMM_Transmit();

COMM_Receive();

COMM_End();

COMM_Init();

COMM_Close();

COMM_Open();

COMM_Transmit();

COMM_Receive();

COMM_End();

COMM_Init();

COMM_Close();

COMM_Open();

COMM_Transmit();

COMM_Receive();

COMM_End();
Chapter 12

PC Services

The code in this book was tested on a PC. It was convenient to create a number of services (i.e., functions) to access some of the capabilities of a PC. These services are invoked from the test code and are encapsulated in a file called PC.c. Because industrial PCs are so popular as embedded system platforms, the functions provided in this chapter could be of some use to you. These services assume that you are running under DOS or a DOS box under Windows 95/98 or NT. You should note that under Windows 95/98 or NT, you have an emulated DOS and not an actual one (i.e., a Virtual x86 session). The behavior of some functions may be altered because of this.

The files PC.c (Listing 12.3) and PC.h (Listing 12.4) are found in the LIBRARY/VIDEO/PC/VS4/85 directory. Unlike the first edition of ESBB, I decided to encapsulate these functions (as they should have been) to avoid defining them in the example code and also, to allow you to easily adapt the code to a different compiler. PC.c basically contains three types of services: character based display, elapsed time measurement, and miscellaneous. All functions start with the prefix PC_

12.00 Character Based Display

PC.c provides services to display ASCII (and special) characters on a PC's VGA display. In normal mode (i.e., character mode), a PC's display can hold up to 2000 characters organized as 25 rows (i.e., y) by 80 columns (i.e., x) as shown in Figure 12.1. Please disregard the aspect ratio of the figure. The actual aspect ratio of a monitor is generally 4 x 3. Video memory on a PC is memory swapped and, on a VGA monitor, video memory starts a absolute memory location 0x00000000 (or using a segment:offset notation, 0:00000000).
Each displayable character requires two bytes to display. The first byte (lowest memory location) is the character that you want to display while the second byte (next memory location) is an attribute that determines the foreground/background color combination of the character. The foreground color is specified in the lower 4 bits of the attribute while the background color appears in bits 4 to 6. Finally, the most-significant bit determines whether the character will blink (when 1) or not (when 0). The character and attribute bytes are shown in Figure 12.2.
Table 12.1 shows the possible colors that can be obtained from the PC's VGA character mode. You will note that you can only have 8 possible background colors but a choice of 16 foreground colors. PC II contains #defines which allow you to select the proper combination of foreground and background colors. These #defines are shown in Table 12.1. For example, to obtain a non-slinking WHITE character on a BLACK background, you would simply add DISP_POND_WHITE and DISP_POND_BLACK (POND means foreground and BOND is background). This corresponds to a HEX value of 0x07 which happens to be the default video attribute of a displayable character on a PC. You should note that because DISP_POND_BLACK has a value of 0x05, you don't actually need to specify it and thus, the attribute for the same WHITE character could just as well have been specified as DISP_POND_WHITE. You should use the #define constants instead of the HEX values to make your code more readable.

The display functions in PC.C are used to write ASCII (and special) characters anywhere on the screen using x and y coordinates. The coordinate system of the display is shown in Figure 12.1. You should note that position 0,0 is located at the upper left corner as opposed to the bottom left corner as you may have expected. This makes the computation of the location of each character to display easier to determine. The address in video memory for any character on the screen is given by:

\[
\text{Address of Character} = 0x00080000 + Y \times 160 + X \times 2
\]

The address of the attribute byte is at the next memory location or:

\[
\text{Address of Attribute} = 0x00080000 + Y \times 160 + X \times 2 + 1
\]

The display functions provided in PC.C perform direct writes to video RAM even though BIOS (Basic Input Output System) services in most PCs can do the same thing but in a portable fashion. I chose to write directly to video memory for performance reasons.

PC.C contains the following five functions which are further described in the interface section of this chapter.
<table>
<thead>
<tr>
<th>Blink</th>
<th>HEX</th>
<th>Color</th>
<th>(B7)</th>
<th>(B6 B5 B4)</th>
<th>Color</th>
<th>(B3 B2 B1 B0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>0x00</td>
<td>Black</td>
<td>0x00</td>
<td></td>
<td>Black</td>
<td>0x00</td>
</tr>
<tr>
<td>Yes</td>
<td>0x80</td>
<td>Blue</td>
<td>0x10</td>
<td></td>
<td>Blue</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Green</td>
<td>0x30</td>
<td></td>
<td>Green</td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cyan</td>
<td>0x30</td>
<td></td>
<td>Cyan</td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Red</td>
<td>0x30</td>
<td></td>
<td>Red</td>
<td>0x04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purple</td>
<td>0x50</td>
<td></td>
<td>Purple</td>
<td>0x05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Brown</td>
<td>0x50</td>
<td></td>
<td>Brown</td>
<td>0x06</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Brown</td>
<td>0x70</td>
<td></td>
<td>Light Brown</td>
<td>0x07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Gray</td>
<td>0x70</td>
<td></td>
<td>Light Gray</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dark Gray</td>
<td>0x70</td>
<td></td>
<td>Dark Gray</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gray</td>
<td>0x70</td>
<td></td>
<td>Gray</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Black</td>
<td>0x70</td>
<td></td>
<td>Light Black</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Cyan</td>
<td>0x70</td>
<td></td>
<td>Light Cyan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Red</td>
<td>0x70</td>
<td></td>
<td>Light Red</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Light Green</td>
<td>0x70</td>
<td></td>
<td>Light Green</td>
<td></td>
</tr>
</tbody>
</table>

### 12.01 Saving and Restoring DOS's Context

The current DOS environment is saved by calling `PC_DOS_SaveContext()` (see Listing 12.1) and would be called by `multitask()`. Then:

1. Setup `uCOS-II`'s context switch vector
2. Setup the tick ISR vector
3. Save DOS's context so that we can return back to DOS when we need to terminate execution of a `uCOS-II` based application.
A lot happens in PC_DOSsaveReturn() so you may need to look at the code in Listing 12.1 to follow along. PC_DOSsaveReturn() starts by setting the flag PC_bExitFlag to FALSE [L.12.1(1)] indicating that we are not returning to DOS. Then, PC_DOSsaveReturn() initializes OSTickDelta to 8 [L.12.1(2)] because this variable will be decremented in OSTickISR(). A value of 0 would have avoided this value to wrap around to 255 when decremented by OSTickISR(). PC_DOSsaveReturn() then sets DOS's tick handler in a free vector table [L.12.1(3)-6(6)] so it can be called by pDOS's tick handler (this is called chaining the vectors). Next, PC_DOSsaveReturn() calls set jmp() [L.12.1(7)], which captures the state of the processor (i.e., the contents of all important registers) into a structure called PC_jmpbuf. Capturing the processor's context will allow us to return to PC_DOSsaveReturn() and execute the code immediately following the call to set jmp(). Because PC_bExitFlag was initialized to FALSE [L.12.1(4)], PC_DOSsaveReturn() skips the code in the if statement [i.e., L.12.1(6)-9(6)] and returns to the caller (i.e., main()).

When you want to return to DOS, all you have to do is call PC_DOSReturn() (see Listing 12.3) which sets PC_bExitFlag to TRUE [L.12.2(1)] and execute a long jmp [L.12.2(2)]. This brings the processor back in PC_DOSsaveReturn() (just after the call to set jmp()) [L.12.1(7)]. This time, however, PC_bExitFlag is TRUE and the code following the if statement is executed. PC_DOSsaveReturn() changes the tick rate back to 18.2 Hz [L.12.8(6)], restores the PC's tick ISR handler [L.12.1(7)], clears the screen [L.12.1(8)], and returns to the DOS prompt through the exit(0) function [L.12.1(9)].

### Listing 12.1 Saving the DOS environment.

```c
void PC_DOSsaveReturn(void) {
    PC_bExitFlag = FALSE; \(1\)
    OSTickDelta = 8; \(2\)
    PC_TICKISR = PC_VectStruct(VEC_TICK); \(3\)
    OSTickISR_CRITICAL(); \(4\)
    set jmp(PC_jmpbuf); \(5\)
    if (PC_bExitFlag == TRUE) { \(6\)
        OSTickISR_CRITICAL(); \(7\)
        PC_TickDelta(10); \(8\)
        PC_VectStruct(VEC_TICK, PC_TICKISR); \(9\)
        OSTickISR_CRITICAL(); \(10\)
        PC_Dispatcher(DISP_PGM_WRITE + DISP_PGM_WRITE); \(11\)
        exit(0); \(12\)
    }
}
```
Listing 12.2 Setting up to return to DOS.

```c
void PC_DOSReturn (void)
{
    PC_ExitFlag = TRUE;
    loop5:(PC_JumpBuffer, 1);
}
```

12.02 Elapsed Time Measurement

The elapsed time measurement functions are used to determine how much time a function takes to execute. Time measurement is performed by using the PC's 8254 timer #2. You make time measurement by wrapping the code to measure by the two functions PC_ElapsedStart() and PC_ElapsedStop(). However, before you can use these two functions, you need to call the function PC_ElapsedInit(). PC_ElapsedInit() basically computes the overhead associated with the other two functions. This way, the execution time (in microseconds) returned by PC_ElapsedStop() consists exclusively of the code you are measuring. Note that none of these functions are reentrant and thus, you must be careful that you do not invoke them from multiple tasks at the same time.

12.03 Miscellaneous

PC_GetDateTime() is a function that obtains the PC's current date and time, and formats this information into an ASCII string. The format is:

```
YYYY-MM-DD  HH:MM:SS
```

and you will need at least 21 characters (including the NUL character) to hold this string. You should note that there are 2 spaces between the date and the time which explains why you need 21 characters instead of 20. PC_GetDateTime() uses the Borland C/C++ library functions gettime() and getdate(), which should have their equivalent on other DOS compilers.

PC_GetKey() is a function that checks for a key that was pressed and if so, obtains that key, and returns it to the caller. PC_GetKey() uses the Borland C/C++ library functions kbdhit() and getch() which again, have their equivalent on other DOS compilers.

PC_GetTickCount() allows you to change the tick rate for pcDOS-II by specifying the desired frequency. Under DOS, the clock occurs 18,26448 times per second or, every 54.925 ms. This is because the 8254 chip used didn't get its counter initialized and the default value of 65553 takes effect. Had the chip been initialized with a divide by 59655, the tick rate would have been a very nice 20,000 Hz! I decided to change the tick rate to something more "exciting" and thus, decided to use about 200 Hz (actually 199.9866). The code found in OS_PCPU.C calls the DOS tick handler one time out of 11. This is done to ensure that some of the housekeeping needed in DOS is maintained. You would not need to do this if you were to set the tick rate to 20 Hz. Before returning to DOS, PC_GetTickCount() is called by specifying 18 as the desired frequency. PC_GetTickCount() will know that you actually mean 18.2 Hz and will correctly set the 8254.
The last two functions in PC.C are used to get and set an interrupt vector. PC_VectGet() and PC_VectSet() should be compiler independent as long as the compiler support the macros %PC_PP() (make far pointer), PP_OFF() (get the offset portion of a far pointer) and PP_SEG() (get the segment of a far pointer).

12.04 Interface Functions

This section provides a reference section for the PC services.
**PC_DispChar()**

```c
void PC_DispChar(int x, int y, char c, int color);
```

`PC_DispChar()` allows you to display a single ASCII (or special) character anywhere on the display.

**Arguments**

- `x` and `y` specify the coordinates (col, row) where the character will appear. Rows (i.e., lines) are numbered from 0 to `DISP_MAX_Y - 1`, and columns are numbered from 0 to `DISP_MAX_X - 1` (see Listing 12.3, PC-12).
- `c` is the character to display. You can specify any ASCII characters and special characters if `c` has a value higher than 128. You can see what characters (i.e., symbols) will be displayed based on the value of `c` by running the test code provided in this book as follows:

```
C:\SOFTWARE\BLOOM\SAMPLE\TEST > TEST display
```

- `color` specifies the contents of the attribute byte and thus the color combination of the character to be displayed. You can add one `DISP_FONT_???` (see Listing 12.4, PC-10) and one `DISP_FONT_???` (see Listing 12.4, PC-10) to obtain the desired color combination.

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void task (void *pdata)
{
    
    for (i=0; i<nrow; i++)
        PC_DispChar(0, 0, '6', DISP_FONT_WHITE);

}
```
PC_DispClrCol()

void PC_DispClrCol(unsigned x, unsigned color);

PC_DispClrCol() allows you to clear the contents of a column (all 25 characters).

Arguments

x specifies which column will be cleared. Columns are numbered from 0 to DISP_MAX_X - 1 (see Listing 12.3, PC.c).

color specifies the contents of the attribute byte. Because the character used to clear a column is the space character (i.e., ' '), only the background color will appear. You can thus specify any of the DISP_KIND_?? colors.

Return Value

None

Notes/Warnings

None

Example

void task (void *data)
{
  for (;;) {
    PC_DispClrCol(0, DISP_KIND_BLACK);
  }
}
PC_DispClrRow()

void PC_DispClrRow(int y, int32_t color);

PC_DispClrRow() allows you to clear the contents of a row (all 80 characters).

Arguments

y specifies which row (i.e., line) will be cleared. Rows are numbered from 0 to DISP_MAX_Y - 1 (see Listing 12.3, PC.C).

color specifies the contents of the attribute byte. Because the character used to clear a row is the space character (i.e., ' '), only the background color will appear. You can thus specify any of the DISP_BKGD_??_colors.

Return Value

None

Notes/Warnings

None

Example

void Tank (void *data)
{
    
    for (i1) {
        
        PC_DispClrRow(10, DISP_BKGD_BLACK);
    
    }
}
PC_DispClrScr()

void PC_DispClrScr(EDRGB color);

PC_DispClrScr() allows you to clear the entire display.

Arguments

color specifies the contents of the attribute byte. Because the character used to clear the screen is the space character (i.e., ' '), only the background color will appear. You can thus specify any of the DISP_BKND_?? colors.

Return Value

None

Notes/Warnings

You should use DISP_FGDND_WHITE instead of DISP_BKND_BLACK because you don’t want to leave the attribute field with black on black.

Example

void Task (void *pdata) 
{
    
    PC_DispClrScr(DISP_FGDND_WHITE);
    for (;;) {
        
    }
}
PC_DispStr() allows you to display an ASCII string. In fact, you could display an array containing any of 255 characters as long as the array itself is NULL terminated.

Arguments

\( x \) and \( y \) specifies the coordinates (col, row) where the first character will appear; rows (i.e., lines) are numbered from 0 to DISP_NTC_Y - 1, and columns are numbered from 0 to DISP_NTC_X - 1 (see Listing 123, PC.C).

\( c \) is a pointer to the array of characters to display. The array must be NULL terminated. Note that you can display any characters from 0x01 to 0xFF. You can see what characters (i.e., symbols) will be displayed based on the value of \( c \) by running the test code provided in this book as follows:

```
C:\SOFTWARE\BLOCKS\SAMPLE\TEST > TEST dispcl
```

color specifies the contents of the attribute byte and thus the color combination of the characters to be displayed. You can add one DISP_BKGND_??? (see Listing 124, PC.C) and one DISP_FGND_??? (see Listing 124, PC.C) to obtain the desired color combination.

Return Value

None

Notes/Warnings

All the characters of the string or array will be displayed with the same color attributes.

Example 81

The code below displays the current value of a global variable called Temperature. The color used depends on whether the temperature is below 100 (white), above 200 (yellow) or if it exceeds 200 (blinking white on a red background).

```
void task(void *data)
{
    char *s[20];

    PC_DispStr(10, 0, "Temperature", DISP_FGND_YELLOW + DISP_BKGND_RED);
```
Example 2

The code below displays a square box 10 characters wide by 7 characters high in the center of the screen.

```c
for (i = 1; i < 7; i++) {
    sprite[10, 1, Temperature];
    if (Temperature < 100.0) {
        color = DISP_PINK_WHITE;
    } else if (Temperature < 200.0) {
        color = DISP_PINK_VIOLET;
    } else {
        color = DISP_PINK_WHITE + DISP_PINK_RED + DISP_BLUE;
        PC_Display(13, 0, a, color);
    }
}
```

```c
if (row[7][11] == {
})
{
    char [10] data[7];
    for (i = 0; i < 10; i++) {
        PC_Display(35, i + 9, row[i], DISP_PINK_WHITE);
    }
}
```
**PC_DOSReturn()**

void PC_DOSReturn(void);

`PC_DOSReturn()` allows your application to return back to DOS. It is assumed that you have previously called `PC_DOSSaveReturn()` in order to save the processor's important registers in order to properly return to DOS. See section 12.01 for a description on how to use this function.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

You must have called `PC_DOSSaveReturn()` prior to calling `PC_DOSReturn()`.

**Example**

```c
void Task (void *pvctx)
{

  INT16U key;

  for (;;) {

    if (PC_DecKey(&key, --TRUE)) {
      if (key == 0x10) {
        PC_DOSReturn(); /* return to DOS */
      }
    }
  }
}
```
PC_DOSSaveReturn()

PC_DOSSaveReturn() allows your application to save the processor’s important registers in order to properly return to DOS before you actually start multitasking with µC/OS-II. You would normally call this function from main() as shown in the example code provided below.

Arguments
None

Return Value
None

Notes/Warnings
You must call this function prior to setting µC/OS-II’s context switch vector (as shown below).

Example

```c
void main (void)
{
    OSInit(); /* Initialize µC/OS-II */
    PC_DOSSaveReturn(); /* Save DOS’ environment */
    pL_VectSet(OSCO, 0); OSCTXSW; /* µC/OS-II’s context switch vector */
    OSFTaskCreate(L1);
    .
    OSStart(); /* Start multitasking */
}
```
**PC_ElapsedInit()**

```c
void PC_ElapsedInit(void);
```

The `PC_ElapsedInit()` function is invoked to compute the overhead associated with the `PC_ElapsedStart()` and `PC_ElapsedStop()` calls. This allows `PC_ElapsedStop()` to return the execution time in microseconds of the code you are trying to measure.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

You must call this function prior to calling either `PC_ElapsedStart()` and `PC_ElapsedStop()`.

**Example**

```c
void main (void)
{
    OSInit();  /* Initialize uC/OS-II */
    ...
    PC_ElapsedInit();  /* Compute overhead of elapsed meas. */
    ...
    OSStart();  /* Start multitasking */
}
```
**PC_ElapsedStart()**

```c
void PC_ElapsedStart(void);
```

**PC_ElapsedStart()** is used in conjunction with **PC_ElapsedStop()** to measure the execution time of some of your application code.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

You must call **PC_ElapsedInit()** before you use either **PC_ElapsedStart()** and **PC_ElapsedStop()**. This function is non-reentrant and cannot be called by multiple tasks without proper protection mechanisms (i.e., semaphores, locking the scheduler, etc.).

The execution time of your code must be less than 54.93 milliseconds in order for the elapsed time measurement functions to work properly.
Example

```c
void main (void)
{
    OSInit(); /* Initialize uC/OS-II */
    
    PC_ElapsedInit(); /* Compute overhead of elapsed time */
    
    OSStart(); /* Start multitasking */
}

void Task (void *pData)
{
    INT16U time_us;
    
    for (;;) {
        
        PC_ElapsedStart(); /* Code you want to measure the execution time */
        time_us = PC_ElapsedStop();
        
    }
}
```
Chapter 12: PC Services — 513

**PC_ElapsedStop()**

`#define PC_ElapsedStop(void)`

PC_ElapsedStop() is used in conjunction with PC_ElapsedStart() to measure the execution time of some of your application code.

**Arguments**

None

**Return Value**

The execution time of your code that was swapped between PC_ElapsedStart() and PC_ElapsedStop(). The execution time is returned in microseconds.

**Notes/Warnings**

You must call PC_ElapsedInit() before you use either PC_ElapsedStart() and PC_ElapsedStop().

This function is non-reentrant and cannot be called by multiple tasks without proper protection mechanisms (i.e., semaphores, locking the scheduler, etc.).

The execution time of your code must be less than 5493 milliseconds in order for the elapsed time measurement functions to work properly.

**Example**

See PC_ElapsedStart() on page 511.
PC_GetDateTime()

void PC_GetDateTime(char *s);

PC_GetDateTime() is used to obtain the current date and time from the PC's real-time clock chip and return this information in an ASCII string that can hold at least 19 characters.

Arguments

*s is a pointer to the storage area where the ASCII string will be deposited. The format of the ASCII string is:

"YYYY-MM-DD HH:MM:SS"

and requires 21 bytes of storage (note that there is 2 spaces between the date and the time).

Return Value

None

Notes/Warnings

None

Example

void Task (void *pdata)
{
    char s[80];

    for (;;) {
        PC_ClearBuffer();
        PC_GetDateTime(s[0]);
        PC_DispStr(0, 24, s, DISP_FONT_WHITE);
    }
}
PC_GetKey()

PC_GetKey() is used to see if a key was pressed at the PC’s keyboard and if so, obtain the value of the key pressed. You would normally invoke this function every so often (i.e., poll the keyboard) to see if a key was pressed. Note that the PC actually obtains key presses through an ISR and buffers key presses. Up to 50 keys are buffered by the PC.

Arguments

key is a pointer to where the key value will be stored. If no key has been pressed, the value will contain 0x00.

Return Value

TRUE is a key was pressed and FALSE otherwise.

Notes/Warnings

None

Example

```c
void Task (void *pdata) {
    INT16U key;
    BOOLEAN avail;

    for (; ; ) {
        avail = PC_GetKey(&key);
        if (avail == TRUE) {
            /* Process key pressed */
            ...
        }
    }
}
```
**PC_SetTickRate()**

```c
void PC_SetTickRate(int freq);
```

`PC_SetTickRate()` is used to change the PC's tick rate from the standard 18.20648 Hz to something faster. A tick rate of 200 Hz is a multiple of 18.20648 Hz (the multiple is 11).

**Arguments**

freq is the desired frequency of the ticker.

**Return Value**

None

**Notes/Warnings**

You can only make the ticker faster than 18.20648 Hz. The higher the frequency, the more overhead you will impose on the CPU. You will have to change `STICKIER()` in order to account for the increased rate (see MicroC/OS-II, The Real-Time Kernel, RAD Books, ISBN 0-87930-543-6).

```c
void Task (void *pxv);
{

  os_ENTER_CRITICAL();
  PC_VectSet(OS06, OS0tickISR);
  PC_SetTickRate(400);  /* Reprogram PC's tick rate to 400 Hz */
  os_EXIT_CRITICAL();

  for (;;) {
    /* ... */
  }
}
```
PC_VectGet ()

void *PC_VectGet(unsigned vect);

PC_VectGet () is used to obtain the address of the interrupt handler specified by the interrupt vector number. As 80x86 processors supports up to 256 interrupt/exception handlers.

Arguments

*vec is the interrupt vector number, a number between 0 and 255.

Return Value

The address of the current interrupt/exception handler for the specified interrupt vector number.

Notes/Warnings

Vector number 0 corresponds to the RESET handler.

It is assumed that the 80x86 code is compiled using the 'large model' option and thus all pointers returned are 'big pointers'.

It is assumed that the 80x86 is running in 'real mode'.

Example

```c
void *p_int_vec;  /* caller must allocate memory for this variable. */

p_int_vec = PC_VectGet(0);  /* get vector handler address */
```
PC_VectSet()

`void PC_VectSet(ESPN vect, void *p user (void));`

`PC_VectSet()` is used to set the contents of an interrupt vector table location. An 80x86 processor supports up to 256 interrupt/exception handlers.

**Arguments**

`vect` is the interrupt vector number, a number between 0 and 255.

`p user` is the address of the interrupt/exception handler.

**Return Value**

None

**Notes/Warnings**

You should be careful when setting interrupt vectors. Some interrupt vectors are used by the operating system (DOS and/or μC/OS-II).

It is assumed that the 80x86 code is compiled using the 'large model' option and thus all pointers returned are 'far pointers'.

If your interrupt handler works in conjunction with μC/OS-II, it must follow the rules imposed by μC/OS-II (see page 91 of *MicroC/OS-II, The Real-Time Kernel*, ISBN 0-87930-543-6).

**Example**

```c
void InterruptHandler (void) {
}

void Task (void *data) {
    
    PC_VectSet(64, InterruptHandler);
    
    for (i = 0; i < 100; i++) {
        
    }
}```
12.05 Bibliography

Chappell, Geoff
DOS Internals
Reading, Massachusetts
Addison-Wesley, 1994
ISBN 0-201-66835-4

Tischer, Michael
PC Intern, System Programming, 5th Edition
Grand Rapids, Michigan
Abrasas, 1995
ISBN 1-55755-282-7

Villani, Pat
FreedOS Kernel, An MS-DOS Emulator for Platform Independence & Embedded Systems Development
Lawrence, Kansas
R&D Books, 1996
ISBN 0-87930-496-7
Listing 12.3  PC.C

-PC SUPPORT FUNCTIONS-
- (c) Copyright 1982-1989, Jan J. Lacobus, Weston, FL
- All Rights Reserved
-* File: PC.C
- By: Jan J. Lacobus

-*****************************************
-#include "includes.h"
-*****************************************
-CONTINUE
-*****************************************

#define DISP_BASEX 0x0800 /* base segment of display (0x0800:EGA, 0x0D00:ATmns) */
#define DISP_NBEX 16 /* maximum number of columns */
#define DISP_NBEY 25 /* maximum number of rows */

#define TIOC_TIM7_G2 0x48 /* 8254 TIM 7 Timer 2 Control Word address */
#define TIOC_TIM7_G1 0x49 /* 8254 TIM 7 Timer 1 Control Word address */
#define TIOC_TIM7_G0 0x4A /* 8254 TIM 7 Timer 0 Control Word address */

#define TIOC_TIM6_G2 0x4B /* 8254 TIM 6 Timer 2 Control Word address */
#define TIOC_TIM6_G1 0x4C /* 8254 TIM 6 Timer 1 Control Word address */
#define TIOC_TIM6_G0 0x4D /* 8254 TIM 6 Timer 0 Control Word address */

#define TIOC_TIM5_G2 0x4E /* 8254 TIM 5 Timer 2 Control Word address */
#define TIOC_TIM5_G1 0x4F /* 8254 TIM 5 Timer 1 Control Word address */
#define TIOC_TIM5_G0 0x50 /* 8254 TIM 5 Timer 0 Control Word address */

#define VECT_IWCK 0x22 /* Vector number for 8254 timer tick */
#define VECT_PW unlock 0x41 /* Vector number used to clear EES */

-*****************************************
-LOCAL GLOBAL VARIABLS
-*****************************************

static int8U PC_BaseAddress;
static int8U PC_BaseAddress;
static int8U PC_BaseAddress;
void (*PC_Init)(void);

/*PAGE*/
Listing 12.3 (continued)  PC.C

/*  
* DESCRIPTION: This function places a single character at a position on the X86 screen. This function 
* draws directly to video RAM instead of using the BIOS for speed reasons. It assumes 
* that the video adapter is VESA compatible. Video RAM seeds at address $03E8 and 
* contains $0000. Each character on the screen is composed of two bytes: the ASCII character 
* to appear on the screen followed by a video attribute. An attribute of $0F7 displays 
* the character on NOTE with a green background. 
* 
* ARGUMENTS: x - X coordinate, y - Y coordinate. 
* 
* RETURNS: None. 
*/

void PCLoOpChar (unsigned x, unsigned y, unsigned c, unsigned attributes)
{
    unsigned offset;

    offset = (((y * HRES + 1) * 2) + (x * VRES + 2)); // Calculate position on the screen.

    putc(attributes + c, video_ram); // Put character in video RAM.
    putc(attributes, video_ram); // Put video attribute in video RAM.

    return;
}
Listing 12.3 (continued)  PC.C

/*
 * Description: This function clears one of the 80 columns on the PC's screen by directly accessing video
 * RAM instead of using the BIOS. It assumes that the video adapter is VGA compatible.
 * Video RAM starts at double address 0x00200000. Each character on the screen is
 * composed of two bytes: the ASCII character to appear on the screen followed by a video
 * attribute. An attribute of 0x07 displays the character in MOUNT WITH a black background.
 * 
 * Arguments:
 * X corresponds to the desired column to clear. Valid column numbers are from
 * 0 to 79. Column 0 corresponds to the leftmost column.
 * 
 * Color specifies the foreground/background color combination to use
 * (see PC.C for available choices)
 * 
 * Returns: None
 */

void dispClearCol (WORD x, WORD color)
{
    BYTE addr;
    for (addr = 0x00200000 + x * 2; ;)
    {
        outb (addr + 1, ' '); /* Put ' ' character in video RAM */
        outb (addr, color);  /* Put video attribute in video RAM */
        addr += 2;          /* Position on next row */
    }
} /* dispClearCol */
Listing 12.3 (continued)  PC.C

/*
 * Display a single character at X, Y.  x, y (coordinates)
 *
 * Description: This function will allow a single character to appear on the
 *              VDU screen. This function will directly write a character using
 * the ASCII definition of the character on the VDU screen. The ASCII
 *              character associated with an attribute can be set as a display
 *              attribute on the terminal. Attributes are stored in different
 *              memory locations and can be used in the current VDU environment.
 *              The attribute table is used to define which character is
 *              associated with a particular attribute in the current VDU
 *              environment. Each attribute is defined by its corresponding
 *              ASCII character and can be set or unset by its associated
 *              display attribute. The attribute is stored in the display
 *              attribute table in memory.
 *
 * Arguments:  x  corresponds to the desired column on the screen. Valid
 *             column numbers are from 0 to 7. Column 8 corresponds to the
 *             extreme right.
 *             y  corresponds to the desired row on the screen. Valid row
 *             numbers are from 0 to 23. Line 0 corresponds to the toprow.
 *             c  is the ASCII character to display. You can also specify a
 *             character with a more specific value than the ASCII character.
 *             If a specific value is not set, the character will
 *             correspond to the foreground/background color to use see PC.M
 *             for available choices and whether the character will blink or not.
 */

int vdu_putchar (int x, int y, int c, int color) {
    fputc (c, std);  
    return 1;  
}

/*
 * Calculate position on the screen
 */
int getvdupos (int x, int y, char *screen, int offset) {
    char *screen = *screen;  
    return (y * 16 + (x + offset) * 8);  
}

/*
 * Set character in video RAM
 */
void setchar (int x, int y, int c, int color) {
    char *screen = *screen;  
    offset = (x + y * 16);  
    fputc (c, std);  
    return 1;  
}
Listing 12.3 (continued)  FC.C

void *shellDrawColumn (DREF x, DREF color)
{
    int ph, x0, x1, y0, y1;
    if (x < 0) x = 0;
    if (x > 127) x = 127;
    dref ph = *DEP_VIA_16 + x * 2;
    dref x0 = DREF_VIA_16 + x * 2;
    dref x1 = DREF_VIA_16 + x * 2;
    for (y0 = y; y0 < 128; y0++) {
        for (x1 = x0; x1 < 128; x1++) {
            DREF_VIA_16[ph + x1] = color;
        }
    }
}

CLEAN A COLUMN

Description: This function clears one of the 16 columns on the screen by directly accessing video RAM instead of using the FC. It assumes that the video adapter is VGA compatible.

Video RAM starts at absolute address 00000000. Each character on the screen is composed of two bytes: the ASCII character to appear on the screen is followed by a video attribute. An attribute of 0 displays the character in white with a black background.

Arguments:

  x

  corresponds to the desired column to clear. Valid column numbers are from 0 to 79. Column 0 corresponds to the leftmost column.

  color

  specifies the foreground/background color combination to use

  (see FC.8 for available choices)

Returns: None

/*
   void *shellDrawColumn (DREF x, DREF color)
{
    int ph, x0, x1, y0, y1;
    if (x < 0) x = 0;
    if (x > 127) x = 127;
    dref ph = *DEP_VIA_16 + x * 2;
    dref x0 = DREF_VIA_16 + x * 2;
    dref x1 = DREF_VIA_16 + x * 2;
    for (y0 = y; y0 < 128; y0++) {
        for (x1 = x0; x1 < 128; x1++) {
            DREF_VIA_16[ph + x1] = color;
        }
    }
}

*/
Listing 12.3 (continued)  PC.C

```
/*
  * Description: This function clears one of the 25 lines on the PC's
  * screen by directly setting the video RAM. It assumes that the video
  * card is compatible with the EGA. Each character on the screen is
  * composed of two bytes: the ASCII character to appear on the screen
  * followed by an attribute byte that displays the character in EGA
  * with a black background.
  * Arguments:  y     corresponds to the desired row to clear. Valid row
  *             numbers are from 0 to 24. Row 0 corresponds to the top
  *             line.
  *             color specifies the foreground/background color
  *             combination to use
  *             (see PC.C for available choices)
  *
  * Returns:     None
  */

void VideoClearRow (int y, int color) {
    int i;
    for (i = 0; i < DISP_WIDTH; i++) {
        /*print */  /* Put character in video RAM */
        /*print */  /* Put attribute in video RAM */
    }
    "\0\0"
}
```
Listing 12.3 (continued)  PC.C

/* ***********************************************************************
 * Description :  This function clears the PC's screen by directly accessing
 *                video RAM instead of using the BIOS. It assumes that the
 *                video adapter is VGA compatible. Video RAM starts at
 *                absolute address 0x00000000. Each character on the screen
 *                is composed of two bytes: the ASCII character to appear on
 *                the screen followed by a video attribute. An attribute
 *                of 0x07 displays the character in MITS white with a black
 *                background.
 * Arguments :  color specifies the foreground/background color combination
 *               to use
 *               (see PC-H for available choices)
 * Returns :  None
 ***********************************************************************
 */

void PC_ClearScreen (DWORD color)
{
    DWORD far *temp;
    temp = (DWORD far *)0x00000000;

    for (i = 0; i < (SCREEN_MX * SCREEN_MY); i++)
    { /* PC display has 64 columns and 25 lines */
        *temp++ = color; /* Put video attribute in video RAM */
        *temp++ = ' '; /* Put " " character in video RAM */
    }
}

/*END*/
Listing 12.3 (continued)  PC.C

/***************************************************************************/

/* This function writes an ASCII string exactly at the given screen position. */
/* The function writes the string directly to video RAM instead of using the */
/* DDE/W or VIDEONAME functions.  X pixels is the x position and */
/* Y pixels is the y position of the text in video RAM. */

/* If the position is outside the video area, the text is written at the */
/* end of the video area. */

/* The screen area is divided into vertical and horizontal segments */
/* When a segment is visible, the whole segment is written to video RAM. */

/* The segment sizes are determined by the /V and /H parameters. */

/***************************************************************************/

/* Description: This function writes an ASCII string exactly at the PC's */
/* screen. This function writes the string directly to video RAM instead */
/* of using the DDE/W or VIDEONAME functions. The position and */
/* attributes of the text are specified in the arguments. */

/* Arguments: */
/* x: Corresponds to the desired column on the screen. Valid column numbers */
/* are from 1 to 79. Column 0 corresponds to the left column. */
/* y: Corresponds to the desired row on the screen. Valid row numbers */
/* are from 0 to 24. */
/* position: Specifies the position of the text. */
/* videoRam: Pointer to video RAM. */
/* color: Specifies the foreground/background color used by the text. */
/* attribute: Specifies the attribute used by the text. */

/***************************************************************************/

int PC_DrawStr (int x, int y, char *position, char *videoRam, int color, int attribute)
{
    int pos, offset, xPix, yPix, attr, pixelRow, pixelCol; /* Define constants. */
    int minX = 0, maxX = 79, minY = 0, maxY = 24; /* Define constants. */

    /* Check if the position is within the video area. */
    if (x < minX || x > maxX || y < minY || y > maxY)
    {
        return(0); /* Return failure. */
    }

    /* Calculate the offset for the character. */
    offset = x * 8 + y * 79; /* Calculate position of character. */

    /* Get the character's attribute. */
    pixelCol = x % 8;
    pixelRow = y % 8;
    attr = attribute;

    /* Put the character in video RAM. */
    *videoRam = *position; /* Write the character to video RAM. */

    /* Put the attribute in video RAM. */
    *videoRam = *position; /* Write the attribute to video RAM. */

    return(1); /* Return success. */
} /* End of function. */
Listing 12.3 (continued)  PC.C

RIEND TO DOS

* Description: This function returns control back to DOS by doing a 'long jump' back to the saved
* location stored in PC_jumpoff. The saved location was established by the function
* PC_DOSReturn(). After execution of the long jump, execution will resume at the
* line following the 'save jump back in PC_DOSReturn()' line, setting the flag
* PC_PlainFlag to TRUE ensures that the 'IF' statement in PC_DOSReturn() executes.
* Arguments: None
* Return: None

void PC_DOSReturn(void)
{
    PC_DosFlag = FALSE; /* Indicate we are returning to DOS */
    PC_jumpoff, 11; /* Jump back to saved environment */
}/

/* D.R */

/* D.K.S. RELEASE IDENTIFICATION */

* Description: This function sets the location of where we are in DOS so that it can be recovered.
* This allows us to abort multithreading under VAX/11 and return back to DOS as if we had
* never left. When this function is called by 'main', it sets PC_DosFlag to FALSE,
* so that we won't take the 'IF' branch. Instead, the CMX registers are saved in the
* long jump buffer 'PC_jumpoff' and we simply return to the caller. If a 'long jump' is
* performed using the jump buffer then, execution would resume at the 'IF' statement and
* this time, if PC_DosFlag is set to TRUE then we would execute the 'IF' statement and
* restore the DOS environment.
* Arguments: None
* Return: None

void PC_DOSReturn(void)
{
    PC_DosFlag = FALSE; /* Indicate that we are not exiting yet */
    /* Initialize the DOS tick counter */
    /* Setup a timer vector */
    /* Store MS-DOS's ticks to chain */
    if (PC_DosFlag = TRUE) /* Capture where we are in DOS */
        PC_jumpoff, 11; /* See if we are exiting back to DOS */
    PC_jumpoff, 11; /* Restore tick count to 182 eq */
    PC_jumpoff, 11; /* Restore DOS's tick vector */
    /* Clear the display */
    /* Return to DOS */
}/

/* D.R */
Listing 12.3 (continued)  PC.C

/*
 * ELAPSED TIME INITIALIZATION
 * 
 * Description: This function initializes the elapsed time module by determining how long the START and 
 * STOP functions take to execute. In other words, this function calibrates this module 
 * to account for the processing time of the START and STOP functions.
 * 
 * Arguments:  None.
 * 
 * Returns:    None.
 */

void PC_ElapsedInit(void)
{
    PC_ElapsedOverhead = 2;
    PC_ElapsedStart();
    PC_ElapsedStop();
}

/*
 * INITIALIZE PC'S TIMER K2
 * 
 * Description: This function initializes the PC's Timer K2 to be used to measure the time between events. 
 * Timer K2 will be PARking when the function returns.
 * 
 * Arguments:  None.
 * 
 * Returns:    None.
 */

void PC_ElapsedTimer(void)
{
    unsigned int tck0; /* timer 0 clock divider */
    tck0 = CDIV4; /* tck0 = (1/4 * OSC_CLK) */
    /* Setup timer K2 for mode 0 */
    outp(TOL0_CTRL, 0x0000);
    outp(TOL0_DATA, 0x0000);
    outp(TOL1_CTRL, 0x0000);
    outp(TOL1_DATA, 0x0000);
    outp(TOL2_CTRL, 0x0000);
    outp(TOL2_DATA, 0x0000);
    /* Start the timer */
    /*
     * ENDPC*/
}
Listing 12.3 (continued)  PC.C

/*
 * STOP THE PC'S TIMER 2 & DISPLAY ELAPSED TIME
 * Description: This function stops the PC's Timer 2, obtains the elapsed count from when it was
 * started and converts the elapsed count to micro-seconds.
 * Arguments:  None.
 * Returns:   The number of micro-seconds since the timer was last started.
 * Notes:     - The returned time estimates for the processing time of the STBK and STPF functions.
 *            - 54924 represents 549240-16 or 0.833699 which is used to convert timer counts to
 *            micro-seconds. The clock source for the PC's timer 2 is 1.19318 MHz (or 8.333039 UU)
 */

BIT16U PC_ElapsedTime(void)
{
    INT8U data;
    INT8U high;
    INT8U low;
    INT8U tick;

    OE_KINT8U(TICK2H);  // Enable the TICK
    data = (INT8U)읍(TICK2L);  // Load the timer value
    data = (INT8U) cú(TICK2H, TICK2L);  // Read the timer
    high = (INT8U) TICK2H;  // High
    low = (INT8U) TICK2L;  // Low
    tick = (INT8U)((喝水 * 54924) + 16) - PC_Elapsed(kk);  // Compute time at task for operation

    return (INT8U) (LONG) data * 54924 or 16 + PC_Elapsed(kk);  // Increment
}
Listing 12.3 (continued)  PC.C

/*
   **************************************************************************
   *  This function obtains the current date and time from the PC.
   *  **************************************************************************
   *  "Arguments" : a is a pointer to where the ASCII string of the current date and time will be stored.
   *                 You may allocate at least 21 bytes (includes the %s at the end of the return string). The date and time will be formatted as follows:
   *                 "YYYY-MM-DD HH:MM:SS"
   *  "Returns" : None
   **************************************************************************
*/

void PC_GetDate(char *a)
{
    struct tm *today;
    struct tm *now;
    char *stop; /*.
    strftime("%Y-%m-%d %H:%M:%S", now->tm_sec, now->tm_min, now->tm_hour, now->tm_year, now->tm_mon, now->tm_mday, now->tm_wday, "Insert Time");
}
Listing 12.3 (continued)  PC.C

/*
 * Checks and gets keyboard key
 *
 * Description: This function checks to see if a key has been pressed on the keyboard and returns TRUE if
 * so. Also if a key is pressed, the key is read and copied where the argument is pointing to.
 *
 * Arguments: *c  is a pointer to where the read key will be stored.
 *
 * Returns:  TRUE if a key was pressed
 *           FALSE otherwise
 *
 */

BOOLEAN PC_readkey (Out ***c)
{
    if (isnull()) { /* See if a key has been pressed */
        return (TRUE);
    } else { /* Get key pressed */
        *c = neggetch();
        return (TRUE);
    }
    /* No key pressed */
}

/*INDE*/
Listing 12.3 (continued)  PC.C

/*
 * SET THE PC'S TICK FREQUENCY
 * Description: This function is called to change the tick rate of a PC.
 * Arguments: freq is the desired frequency of the clock (in Hz)
 * Returns: none
 *
 * Notes: 1) The magic number 2396360 is actually below the CPU's frequency of the 8254 chip which
 * is always 1.191190 MHz.
 * 2) The equation computes the counts needed to load into the 8254. The strange equation
 * is actually used to round the number using integer arithmetic. This is equivalent to
 * the following prime equation:
 * count * 1193182.0 Hz
 * freq
 * count = 4095.0 / 0.5
 */

void PC_setTickRate (uint32_t freq)
{
    uint32_t counts;

    if (freq == 0) {
        /* See if we need to restore the IC5 frequency */
        count = 0;
    } else if (freq > 0) {
        /* Compute 8254 counts for desired frequency and ... */
        /* ... round to nearest count */
        count = (uint32_t) (((uint32_t)(freq) / 1193182.0) + 0.5);
    } else {
        /* */
    }
Listing 12.3 (continued)  PC.C

/**
 * INSTALL INTERRUPT VECTORS
 *
 * Description: This function sets an interrupt vector in the interrupt vector table.
 * Arguments: vect is the desired interrupt vector number, a number between 1 and 255.
 *            lac is a pointer to a function to execute when the interrupt or exception occurs.
 * Returns: none
 *
 * void PC_VectSet (INTNUM vect, void (*lac)(void))
 * { DINTU space;
 *   vect = ((INT8U) * CRG_FPPTR+1500, vect + 4); /* point into TIP at desired vector location */
 *   CRG_INTR_PICBTRY();
 *   *lac = vect;
 *   CRG_INTR_PICBTRY();
 *   return (OK);
 *  }
 */
Listing 12.4  PC.H

/*
 ** PC.H
 */

/*
 ** Copyright 1982-1999, Jesu J. Labrana, Weston, Fl.
 ** All Rights Reserved
 *
 ** Brief: PC.H
 ** By: Jesu J. Labrana
 */

/*
 **
 ** CONTENTS
 **
 ** COLOR AT&TLIBRARY FOR VDISP DRIVER
 **
 ** Description: These definitions are used in the PC_Display() functions. The 'color' argument in these
 ** functions MIGHT specify a 'foreground' color, a 'background' and whether the display will
 ** blank or not. If you don't specify a background color, BLACK is assumed. You would
 ** specify a color combination as follows:
 **
 ** PC_Display(0, 0, 'A', DESPB PhúNIGHT + DBSP PhúRGB + DBSP PhúG);
 **
 ** To have the ASCII character 'A' blink with a white letter on a blue background.
 **
 **
 **
 **
 **
 */

#define DESPB PhúNIGHT 0x00
#define DESPB PhúRGB 0x01
#define DESPB PhúG 0x02
#define DESPB PhúVIBRANT 0x03
#define DESPB PhúWHITE 0x04
#define DESPB PhúGRAY 0x05
#define DESPB PhúLIGHT_GRAY 0x06
#define DESPB PhúLIGHT_GREEN 0x07
#define DESPB PhúLIGHT_RED 0x08
#define DESPB PhúLIGHT_BLUE 0x09
#define DESPB PhúLIGHT_CYAN 0x0A
#define DESPB PhúLIGHT_PURPLE 0x0B
#define DESPB PhúVIBRANT_3 0x0C
#define DESPB PhúVIBRANT_4 0x0D
#define DESPB PhúVIBRANT_5 0x0E
#define DESPB PhúVIBRANT_6 0x0F
#define DESPB PhúWHITE 0x10
#define DESPB PhúRED 0x11
#define DESPB PhúGREEN 0x12
#define DESPB PhúCYAN 0x13
#define DESPB PhúVEIW 0x14
#define DESPB PhúBEIGE 0x15
#define DESPB PhúMAPLE 0x16
#define DESPB PhúGRAY 0x17
#define DESPB PhúDARK_GRAY 0x18
#define DESPB PhúOlive 0x19
#define DESPB PhúPINE 0x1A
#define DESPB PhúTEAL 0x1B
#define DESPB PhúSAGE 0x1C
#define DESPB PhúNAVY 0x1D
#define DESPB PhúMAROON 0x1E
#define DESPB PhúMAROON_GRAY 0x1F
#define DESPB PhúBLUE 0x20
#define DESPB PhúGREEN 0x21
#define DESPB PhúYELLOW 0x22
#define DESPB PhúPLUM 0x23
#define DESPB PhúPURPLE 0x24
#define DESPB PhúPINK 0x25
#define DESPB PhúWINE 0x26
#define DESPB PhúDARK_PINK 0x27
#define DESPB PhúCARAMEL 0x28
#define DESPB PhúOLIVE 0x29
#define DESPB PhúOLIVE_1 0x2A
#define DESPB PhúOLIVE_2 0x2B
#define DESPB PhúOLIVE_3 0x2C
#define DESPB PhúOLIVE_4 0x2D
#define DESPB PhúOLIVE_5 0x2E
#define DESPB PhúOLIVE_6 0x2F
*/
Listing 12.4 (continued)  \texttt{PC.H}

```c
/*
  * FUNCTION PROTOTYPES
  */

void PC_DrawChar(int x, int y, int width, int color);
void PC_DrawPoint(int x, int y, int color);
void PC_DrawRect(int x, int y, int width, int color);
void PC_DrawLine(int x1, int y1, int x2, int y2, int color);
void PC_Begin(int x, int y, int width, int color);
void PC_End();
void PC_BeginLine(int width);
void PC_EndLine(int width);
void PC_DrawBox(int x, int y, int width, int height, int color);
void PC_ClearScreen(void);
void PC_Clear(int x, int y, int width, int height);
void PC_DrawChar(int x, int y, int width, int color);
void PC_DrawFont(int x, int y, int width, int height, int color);
void PC_BeginFont(int x, int y, int width, int height, int color);
void PC_EndFont();
void PC_DrawChar(int x, int y, int width, int color);
void PC_DrawFont(int x, int y, int width, int height, int color);
void PC_BeginFont(int x, int y, int width, int height, int color);
void PC_EndFont();
*/
```
Appendix A

**μC/OS-II, The Real-Time Kernel**

μC/OS-II is a portable, ROM-able, preemptive, real-time, multitasking kernel that can manage up to 63 tasks. μC/OS-II is comparable in performance to many commercially available kernels. μC/OS-II was written in C with microprocessor-specific code written in assembly language. Assembly language was kept to a minimum so that μC/OS-II can easily be ported to other target microprocessors.

Most modules presented in this book assume that services are provided by a real-time multitasking kernel. Because of this, I have provided, in object form, a scaled down version of μC/OS-II, The Real-Time Kernel v2.00 that will allow you to test all of the code in this book. In other words, only the features needed to run the examples are provided.

The complete source code (along with a port for the Intel 80x86, large model) for μC/OS-II is available in my book: MicroC/OS-II, The Real-Time Kernel (ISBN 0-87930-543-6), also published by R&D Books (see the ad in the back of this book.) The source code for μC/OS-II is available on a floppy diskette (MS-DOS format) which is included with the book. Along with providing the source code for μC/OS-II, the book describes the internals, explains how the kernel works, and allows you to port μC/OS-II to other microprocessors (if needed). You can also obtain port to many processors through the official μC/OS and μC/OS-II web site at www.uCOS-II.com. μC/OS-II provides the following features:

- create and manage up to 63 tasks,
- create and manage a large number of semaphores,
- delay tasks for an integer number of ticks or a user-specified amount of time in hours, minutes, seconds, and milliseconds,
- lock/unlock the scheduler,
- service interrupts,
- allows you to change the priority of tasks,
- lets you delete tasks,
- allows tasks to suspend and resume other tasks,
- manages a large number of message mailboxes and queues for intertask communications,
536 — Embedded Systems Building Blocks, Second Edition

- provides fixed-sized memory block management.
- manages a 32-bit system clock.

Even though Embedded Systems Building Blocks, Second Edition assumes the presence of μC/OS-II, you can easily adapt the code in this book to any other real-time kernel as long as the kernel provides the same services (most other kernels do). If you do not have a real-time kernel, you can easily modify some of the code to work in a foreground/background environment.

The version of μC/OS-II in this book is provided in object form for the Intel 80x86 Large Model and has been compiled with the Bland's C++ v4.51. The compiler was instructed to generate code for any Intel 80x86 which has hardware floating-point support. You can thus use the code on any PC having either an Intel 8086, Pentium, Pentium-II, Pentium-III and processors from AMD which have floating-point hardware.

I configured μC/OS-II to limit the number of tasks to 15 and the number of semaphores to 10. You will not be able to invoke either the queue or memory management feature of μC/OS-II because they have been disabled in OS_CPU.H.

The object code for μC/OS-II is found in the \SOFTWARE\BLOCKS\SAMPLE\OBJ directory in these files:

- uCOS_ITT.OBJ μC/OS-II (compiled from the C source)
- OS_CPU_C.OBJ 80x86 microprocessor specific, large model with hardware floating-point support (compiled from the C source)
- OS_CPU_AV.OBJ 80x86 microprocessor specific (assembled from the ASM source)

You will need to link these files with your application if you are planning on using this version of μC/OS-II.

When you use μC/OS-II, you will need to include the following header files in your source code:

- OS_CPU.H which is found in \SOFTWARE\μC/OS-II\386-PP\C\SOURCE
- UOS_ITT.H which is found in \SOFTWARE\μC/OS-II\SOURCE

You should note that OS_CPU.H must be listed first. Also, you cannot change any of the #defines that are provided in these files. If you do, your application may not work properly. The only way to change the #defines is to obtain the full source code for μC/OS-II (see foreword statement).

I included a μC/OS-II mini-reference section which contains only the functions used in this book.
`OSInit()`

`void OSInit(void);`

`OSInit()` is used to initialize `µC/OS-II`. `OSInit()` must be called prior to calling `OSTart()` which will actually start multitasking.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

`OSInit()` must be called before `OSTart()`.

**Example**

```c
void main (void) {
    
    OSInit();  /* Initialize µC/OS-II */

    OSEvent();  /* Start Multitasking */
}
```
OSSemCreate()  
OS_EVENT *OSSemCreate(DWORD value);

OSSemCreate() is used to create and initialize a semaphore. A semaphore is used to:
1. Allow a task to synchronize with either an ISR or a task
2. Gain exclusive access to a resource
3. Signal the occurrence of an event

Arguments
value is the initial value of the semaphore. The initial value of the semaphore is allowed to be between 0 and 65535.

Return Value
A pointer to the event control block allocated to the semaphore. If no event control block is available, OSSemCreate() will return a NULL pointer.

Notes/Warnings
Semaphores must be created before they are used.

Example

```
void main()
{

    OSInit();  /* Initialize uCOS-II */

    OSInit();  /* Create Display Semaphore */

    OSStart();  /* Start Multitasking */

```
Appendix A: μC/OS-II, The Real-Time Kernel — 539

OSSemPend()

void OSSemPend(OS_EVENT *event, int16_t timeout, int8_t *err);

OSSemPend() is used when a task desires to get exclusive access to a resource, synchronize its activities with an ISR, a task, or until an event occurs. If a task calls OSSemPend() and the value of the semaphore is greater than 0, then OSSemPend() will decrement the semaphore and return to its caller. However, if the value of the semaphore is equal to zero, OSSemPend() places the calling task in the waiting list for the semaphore. The task will then wait until a task or an ISR signals the semaphore or, the specified timeout expires. If the semaphore is signaled before the timeout expires, μC/OS-II will resume the highest priority task that is waiting for the semaphore. A pended task that has been suspended withOSTaskSuspend() can obtain the semaphore. The task will, however, remain suspended until the task is resumed by callingOSTaskResume().

Arguments

prevent is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created (see OSSemCreate() on page 538).

timeout is used to allow the task to resume execution if a message is not received from the mailbox within the specified number of clock ticks. A timeout value of 0 indicates that the task desires to wait forever for the message. The maximum timeout is 65535 clock ticks. The timeout value is not synchronized with the clock tick. The timeout count starts being decremented on the next clock tick which could potentially occur immediately.

err is a pointer to a variable which will be used to hold an error code. OSSemPend() sets *err to either:

1. 0, no error, the semaphore is available.
2. OS_NO_ERR, the semaphore was not signaled within the specified timeout.
3. OS_SRH_PEND_ISR, you called this function from an ISR and μC/OS-II would have to suspend the ISR. In general, you should not call OSSemPend() if μC/OS-II checks for this situation in case you do anyway.

Return Value

None

Notes/Warnings

Semaphores must be created before they are used.
Example

OS_EVENT *DispEsem;

void DispTask(void *pData) {
    INT32 err;

    pData = pData;
    for (;;) {
        .
        OSSesPend(DispEsem, 0, &err);
        /* The only way this task continues is if */
        /* - the semaphore is signaled */
    }
}
**OSSemPost()**

```c
OSSemPost (OS_SEMPOST *psem);
```

A semaphore is signaled by calling OSSemPost(). If the semaphore value is greater than or equal to zero, the semaphore is incremented and OSSemPost() returns to its caller. If tasks are waiting for the semaphore to be signaled then, OSSemPost() removes the highest priority task pending (waiting) for the semaphore from the waiting list and makes this task ready to run. The scheduler is then called to determine if the awakened task is now the highest priority task ready to run.

**Arguments**

- `psem`: a pointer to the semaphore. This pointer is returned to your application when the semaphore is created (see OSSemCreate() on page 538).

**Return Value**

OSSemPost() returns one of these two error codes:

1. OS_SEM_OK, if the semaphore was successfully signaled
2. OS_SEM_OVF, if the semaphore count overflowed

**Notes/Warnings**

Semaphores must be created before they are used.
Example

OS_EVENT *DispData;

void TaskA(void *pdata)
{
    int32 err;

    pdata = pdata;
    for (i;i <
        err = OSEventPost(DispDat);
        if (err == OS_JCR_OK) {
            /* Semaphore signaled */
        } else {
            /* Semaphore has overflowed */
        }
    }
}
OSStart() is used to start multitasking under μC/OS-II.

Arguments
None

Return Value
None

Notes/Warnings
OSInit() must be called prior to calling OSStart(). OSStart() should only be called once by your application code. If you call OSStart() more than once, OSStart() will do nothing on the second and subsequent calls.

Example

```c
void main(void) {
    /* User Code */
    OSInit();  /* Initialize μC/OS-II */
    /* User Code */
    OSStart();  /* Start Multitasking */
}
```
**OSStatInit()**

```c
void OSStatInit(void);
```

OSStatInit() is used to have μC/OS-II determine the maximum value that a 32-bit counter can reach when no other task is executing. This function must be called when there is only one task created in your application and, when multitasking has started. In other words, this function must be called from the first, and only created task.

**Arguments**

None

**Return Value**

None

**Notes/Warnings**

None

**Example**

```c
void FirstAndOnlyTask (void *pdata)
{

  OSStatInit();  /* Compute CPU capacity with no task running */
  OSTaskCreate(); /* Create the other tasks */

  for (;;) {

  }
}
```
OSTaskCreate()

OSTaskCreate() allows an application to create a task so it can be managed by μC/OS-II. Tasks can either be created prior to the start of multitasking or by running a task. A task cannot be created by an ISR. A task must be written as an infinite loop as shown in the example below and must not return.

OSTaskCreate() is used for backward compatibility with μC/OS and when the added features of OSTaskCreateExt() are not needed.

Depending on how the stack frame was built, your task will either have interrupts enabled or disabled. You will need to check with the processor specific code for details.

Arguments

Task is a pointer to the task's code.

pdata is a pointer to an optional data area which can be used to pass parameters to the task when it is created. Where the task is concerned, think it was invoked and passed the argument pdata as follows:

void Task(void *pdata)
{
    /* Do something with pdata */
    for (;;) { /* Task body, always an infinite loop */

        /* Must call one of the following services */
        /* osEventPend() */
        /* osCosPend() */
        /* OSRealPend() */
        /* OSTimeDelay() */
        /* OSRealTimeMsg() */
        /* OSTaskSuspend() */
        /* OS_TaskDel() */
        /* OSEventDel() */
    }
}

ptos is a pointer to the task's top of stack. The stack is used to store local variables, function parameters, return addresses, and CPU registers during an interrupt. The size of the stack is determined by the task's requirements and, the anticipated interrupt testing. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all nested functions, as well as requirements for interrupts (accounting for nesting). If the configuration constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e., from high memory to low
memory). The stack will thus need to point to the highest valid memory location on the stack. If OS_STKDEPTH is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low memory to high memory).

perio is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.

**Return Value**

`OSTaskCreate()` returns one of the following error codes:

1. 0xNO_ERR, if the function was successful
2. 0xPRIO_EXIST, if the requested priority already exist

**Notes/Warnings**
The stack must be declared with the OS_STK type.

A task must always invoke one of the services provided by µC/OS-II to either wait for time to expire, suspend the task or, wait an event to occur (wait on a mailbox, queue, or semaphore). This will allow other tasks to gain control of the CPU.

You should not use task priorities 0, 1, 2, 3 and OS_LOWEST_P prio-3, OS_LOWEST_Prio-2, OS_LOWEST_Prio-1 and OS_LOWEST_Prio because they are reserved for µC/OS-II use. This thus leaves you with up to 56 application tasks.

**Example**

This example shows that the argument that Task[1] will receive is not used and thus, the pointer `psTask` is set to NULL. Note that I assumed that the stack grows from high memory to low memory because I passed the address of the highest valid memory location of the stack `Task[0]`. If the stack grows in the opposite direction for the processor you are using, you will need to pass `Task[0]` as the task’s top-of-stack.
 Appendix A: μC/OS-II, The Real-Time Kernel — 547

void main(void)
{
    int err;
    OSInit();            /* Initialize μC/OS-II */
    OSMemInit();
    OSProcCreate(Task1,
                  (void *)Task1Data,
                  &Task1Stk[1024],
                  256);
    OSEnter();           /* Start Multitasking */
}

void Task1(void *pdata)
{
    pdata = pdata;
    for (;;) {
        /* Task code */
    }
}
OSTaskCreateExt() allows an application to create a task so it can be managed by μC/OS-II. This function serves the same purpose as OSTaskCreate() except that it allows you to specify additional information about your task to μC/OS-II. Tasks can either be created prior to the start of multitasking or by a running task. A task cannot be created by an ISR. A task must be written as an infinite loop as shown in the example code below and, must not return. Depending on how the stack frame was built, your task will either have interrupts enabled or disabled. You will need to check with the processor specific code for details. You should note that the first four arguments are exactly the same as the ones for OSTaskCreate(). This was done to simplify the migration to this new, and more powerful function.

Arguments

task is a pointer to the task’s code.
pdata is a pointer to an optional data area which can be used to pass parameters to the task when it is created. Where the task is concerned, it thinks it was invoked and passed the argument pdata as follows:

```c
void Task (void *pdata)
{
    /* Do something with pdata */
    for (;;) {
        /* Task body, always an infinite loop. */

        /* Must call one of the following services: */
        /* OSTaskPend(); */
        /* OSServePend(); */
        /* OSTimeInfo(); */
        /* OSTimeInfoExt(); */
        /* OSTaskSuspend(); (Suspend self) */
        /* OSTaskDel(); (Delete self) */
    }
}
```

ptos is a pointer to the task’s top of stack. The stack is used to store local variables, function parameters, return address, and CPU registers during an interrupt. The size of the stack is determined by the task’s requirements, and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all
nested functions, as well as requirements for intertasks (accounting for nesting). If the configuration constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e., from high memory to low memory). In this case, the stack will thus need to point to the highest valid memory location on the stack. If OS_STK_GROWTH is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low memory to high memory).

prior is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority (i.e., the importance) of the task.

id is the task’s ID number. At this time, the ID is not currently used in any other function and has simply been added in OSHandleCreateExt() for future expansion. You should set the id to the same value as the task’s priority.

prio is a pointer to the task’s bottom of stack. If the configuration constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward (i.e., from high memory to low memory) and thus, prio must point to the lowest valid stack location. If OS_STK_GROWTH is set to 0, the stack is assumed to grow in the opposite direction (i.e., from low memory to high memory) and thus, prio must point to the highest valid stack location. prio is used by the stack checking function OSStackCheck().

stk_size is used to specify the size of the task’s stack (in number of elements). If OS_STK is set to INT8U, then stk_size corresponds to the number of bytes available on the stack. If OS_STK is set to INT16U, then stk_size contains the number of 16-bit entries available on the stack. Finally, if OS_STK is set to INT32U, then stk_size contains the number of 32-bit entries available on the stack.

pewr is a pointer to a user-specified memory location (typically a data structure) which is used in a TCB extension. For example, this user memory can hold the contents of floating-point registers during a context switch, the time each task takes to execute, the number of times the task is switched-in, etc.

opt contains task-specific options. The lower 8 bits are reserved by pCOS-II but you can use the upper 8 bits for application-specific options. Each option consists of a bit. The option is selected when the bit is set. The current version of pCOS-II supports the following options:

- OS_TASK_OPT_STK_CHK specifies whether stack checking is allowed for the task.
- OS_TASK_OPT_STK_CLR specifies whether the stack needs to be cleared.
- OS_TASK_OPT_SAVE_FP specifies whether floating-point registers will be saved.

You should refer to pCOS-II for other options, i.e., OS_TASK_OPT_???.

Return Value

OSHandleCreateExt() returns one of the following error codes:
1. OS_NO_ERR, if the function was successful
2. OS_PTD0_EXIST, if the requested priority already exists

Notes/Warnings

The stack must be declared with the OS_STK type.

A task must always invoke one of the services provided by pCOS-II to either wait for time to expire, suspend the task, or wait for an event to occur (wait on a mailbox, queue, or semaphore). This will allow other tasks to gain control of the CPU.
You should not use task priorities 0, 1, 2, 3 and OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1 and OS_LOWEST_PRIO because they are reserved for μC/OS-II's use. This thus leaves you with up to 56 application tasks.

Example
The task control block is extended (1) using a 'user defined' data structure called TASK_USER_DATA (2) which, in this case, contains the name of the task as well as other fields. The task name is initialized with the strcpy() standard library function (3). Note that stack checking has been enabled (4) for this task and thus, you are allowed to call OSTaskSleep() (5). Also, we assume here that the stack grown downward (5) on the processor used (i.e., OS_STK_GROWTH is set to 1). TOSS stands for 'Top-Of-Stack' and EOS stands for 'Bottom-Of-Stack'.

typedef struct {
  char TaskName[20];
  INT16U TaskCtr;
  INT16U TaskStartTme;
  INT32U TaskTotExecTime;
} TASK_USER_DATA;

void main(void)
{
  intU8 err;

  OSInit(); /* Initialize μC/OS-II */
  strcpy(TaskUserData.TaskName, "MyTaskName"); /* (3) Name of task */
  err = OSTaskCreate(&Task,
                     (void *)TaskUserData,
                     OS_TASK_DEFAULT_TASK,    /* (1) TCB Extension */
                     OS_STKERRQ.ERR_TASK_DATA, /* (4) Stack checking enabled */
                     &Task); /* Start Multitasking */

  while (1) {
  }
}
void Task(void *pdata) {
  pdata = plate;  /* Avoid compiler warning */
  for (;;) {
    /* Task code */
  }
}
OSTimeDly() allows a task to delay itself for a number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from 0 to 65535 ticks. A delay of 0 means that the task will not be delayed and OSTimeDly() will return immediately to the caller. The actual delay time depends on the tick rate (see OS_TICKS_PER_SEC in the configuration file OS_CFG.h).

Arguments

ticks is the number of clock ticks to delay the current task.

Return Value

None

Notes/Warnings

Note that calling this function with a delay of 0 results in no delay and thus the function returns immediately to the caller. To ensure that a task delays for the specified number of ticks, you should consider using a delay value that is one tick higher. For example, to delay a task for at least 10 ticks, you should specify a value of 11.

Example

```c
void TaskX(void *pdata)
{
    for (;;) {
        .
        .
        OSTimeDly(10); /* Delay task for 10 clock ticks */
    }
}
```
OSTimeDlyHMSM()

void OSTimeDlyHMSM(int32 hours, int32 minutes, int32 seconds, int32 milliseconds);

OSTimeDlyHMSM() allows a task to delay itself for a user-specified amount of time specified in hours, minutes, seconds, and milliseconds. This is a more convenient and natural format than ticks. Rescheduling always occurs when at least one of the parameters is non-zero.

Arguments

hours is the number of hours that the task will be delayed. The valid range of values is from 0 to 255 hours.

minutes is the number of minutes that the task will be delayed. The valid range of values is from 0 to 59.

seconds is the number of seconds that the task will be delayed. The valid range of values is from 0 to 59.

milliseconds is the number of milliseconds that the task will be delayed. The valid range of values is from 0 to 999. Note that the resolution of this argument is in multiples of the tick rate. For instance, if the tick rate is set to 10 ms, then a delay of 5 ms would result in no delay. The delay is actually rounded to the nearest tick. Thus a delay of 15 ms would actually result in a delay of 20 ms.

Return Value

OSTimeDlyHMSM() returns one of the following error codes:

1. OS_NO_ERR, if you specified valid arguments and the call was successful.
2. OS_TIME_INVALID_MINUTES, if the minutes argument is greater than 59.
3. OS_TIME_INVALID_SECONDS, if the seconds argument is greater than 59.
4. OS_TIME_INVALID_MS, if the milliseconds argument is greater than 999.
5. OS_TIME_ZERO_DLY, if all four arguments are 0.

Notes/Warnings

Note that calling this function with a delay of 0 hours, 0 minutes, 0 seconds, and 0 milliseconds results in no delay and thus the function returns immediately to the caller. Also, if the total delay time ends up being larger than 65535 clock ticks then, you will not be able to assert the delay and resume the task by calling OSTimeDlyResume().
Example

```c
void *task(void *pdata)
{
    for (i; ) { 
        OSTimeDlyMicros(0, 0, 1, 0); /* Delay task for 1 second */
    }
}
```
OSVersion()

INT16U OSVersion();

OSVersion() is used to obtain the current version of μC/OS-II.

Arguments
None

Return Value
The version is returned as x.yy multiplied by 100. In other words, version 2.00 is returned as 200.

Notes/Warnings
None

Example

void TaskX(void *pdata)
{
    INT16U os_version;

    for (;;){
        os_version = OSVersion(); /* obtain uC/OS-II's version */
    }
}
**OS_ENTER_CRITICAL()** and **OS_EXIT_CRITICAL()**

**Arguments**
None

**Return Value**
None

**Notes/Warnings**
These macros must be used in pair.

**Example**

```c
void Task() {
    for (;;) {
        OS_ENTER_CRITICAL(); /* Disable interrupts */
        /* Access critical code */
        OS_EXIT_CRITICAL(); /* Enable interrupts */
    }
}
```
Listing A.1  OS_CPU.H

azon: OS_CPU.H

* File : OS_CPU.H
* By : Jan J. LaBrosse
* Version : 1.00

******************************************************************************

/*
 * OS_CPU.H

 */

#define OS_CPU_H

/*
 * DATA TYPES
 */

/* Compiler-specific */

typedef unsigned char BYTE16;
typedef unsigned char BYTE16M;
typedef unsigned long BYTE16L;
typedef unsigned long BYTE16ML;
typedef unsigned long double BYTE16DL;
typedef unsigned int BYTE16I;
typedef unsigned int double BYTE16DI;

#define CODE16 1
#define CODE16M 1
#define CODE16L 1
#define CODE16ML 1
#define CODE16DL 1
#define CPU16 1
#define CPU16M 1
#define CPU16L 1
#define CPU16ML 1
#define CPU16DL 1

#include <stdlib.h>

#pragma pack(1)

#define _CPU16

/******************************************************************************/
Listing A.1 (continued) OS_CPU.H

---

* Define OS_CRITICAL_METHOD to 2

* Define OS_INTR_CRITICAL() asm ret
  * Disable interrupts

* Define OS_INT_RT_CRITICAL() asm ret
  * Disable interrupts

* Define OS_INTR_CRITICAL() asm ret
  * Disable interrupts

* Define OS_INT_RT_CRITICAL() asm ret
  * Disable interrupts

---

* Define OS_STACK_DEPTH = 1
  * Stack grows from HIGH to LOW memory on $80H

* Define uOSRetVal 0x0
  * Interrupt vector # used for context switch

* Define OS_TASK_SW() as INT uOSRetVal

* GLOBAL vOS_CRITICAL

* Define OS_CPU_TASK_GETTick() as INT 0
  * Counter used to invoke uOS’s tick handler every ‘n’ ticks

---

* FUNCTION void OS_EXIT_CRITICAL()

* void OS_ENTER_CRITICAL();
  * void OS_EXIT_CRITICAL();
* void OS_ENTER_CRITICAL();
  * void OS_EXIT_CRITICAL();
Listing A.2  uCOS_II.H

/**
 *********************************************************
 * uCOS-II
 * The Real-Time Kernel
 * (c) Copyright 1999, Jean J. Labrosse, Worton, MI.
 * All Rights Reserved
 *
 * File : uCOS_II.h
 * By  : Jean J. Labrosse
 *********************************************************
 */

/*UEND*/

Listing A.2 (continued) ucos_ii.h

/**
 *---------------------------------------------------------------
 * EVENT CONTROL BLOCK
 *
 *---------------------------------------------------------------
 */

#if (UC_MAX_EVENTS == 2)

typedef struct {
    void *OSMEMRdy;
    int8U OSEvent[NUM_OS_EVENT_TYPE_SIZE];
    int8U OSEventCount;
    int8U OSEventType;
} OSEventType;
#endif

#if UC_TASKS

typedef struct {
    OSEvent;
} OSEvent;
#endif

#if UC_MAX_MAIL

typedef struct {
    OSEvent;
} OSEventMail;
#endif

#if UC_MEM_PARTITION == 2

typedef struct {
    OSEvent;
} OSEventMem;
#endif

void *OSMEMRdy;
void OSEventMem;
void OSEventMail;

#endif

#if UC_MAX_EVENTS

#endif

*/

/*GMEM */
Listing A.2 (continued)  uCOS_HH

/*
++++++++++++++++++++++++++++++++++++++++++++++++++>
MESSAGE QUEUE DATA
+++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*
*/
#define ULQ_WM ( */ /* Pointer to next message to be extracted from queue */ /*
INT16U WMIndex; /* Number of messages in message queue */ /*
INT16U QMax; /* Size of message queue */ /*
INT16U EffectiveULQ_WM_Pointer; /* List of tasks waiting for event to occur */ /*
INT16U LQChildren; /* Group corresponding to tasks waiting for event to occur */ /*
) ULQ_WM_DATA;
#endif
*/
*/
/*
++++++++++++++++++++++++++++++++++++++++++++++++++>
SEMIPHERE DATA
+++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*
*/
#define ULQ_WM_WM ( */ /* Semaphore count */ /*
INT16U WMCount; /* List of tasks waiting for event to occur */ /*
INT16U WMChildren; /* Group corresponding to tasks waiting for event to occur */ /*
) ULQ_WM_WM_DATA;
#endif
*/
*/
/*
++++++++++++++++++++++++++++++++++++++++++++++++++>
TASK STACK DATA
+++++++++++++++++++++++++++++++++++++++++++++++++++*/
/*
*/
#define ULSTACK_WM ( */ /* Number of free bytes on the stack */ /*
INT16U WMFree; /* Number of bytes used on the stack */ /*
) ULSTACK_WM_DATA;
#endif
*/
/*#define */
Listing A.2 (continued) uCOS-II.H


typedef struct os_tcb {
    U32TH  OS_RTN;  /* Pointer to current top of stack */
    void * OS_CTX;  /* Pointer to user definable data for TCB extension */
    OS_HWK OS_CTXBLOCK; /* Pointer to bottom of stack */
    OS_SIZE OS_CTXSIZE; /* Size of task stack (in number of stack elements) */
    OSSizeT OS_CTXOPT; /* Task options as passed by OSInitBackend() */
    OSSizeT OS_CTXOPT; /* Task opts in (0..65535) */
} TCB;

/*
 * OS_tcb, OS_tcbNext;
 * Pointer to next TCB in the TCB list
 */

void OS_tcbNext;

/*
 * OS_tcbPrevious;
 * Pointer to previous TCB in the TCB list
 */

OS_tcbPrevious;

/*
 * OS_tcb; /* Pointer to event control block */
 */

void OS_tcb;

/*
 * OS_tcbInit;
 * Message received from OSKernelPost() or OSKernelPost() */
 */

OS_tcbInit;

/*
 * OS_tcbSetError;
 * Exit codes: 0 = SUCCESS, -1 = ERROR */
 */

OS_tcbSetError;

/*
 * OS_tcbReset; /* Indicate whether a task needs to delete itself */
 */

OS_tcbReset;

/*
 */

="/SPACE"
Listing A.2 (continued) uCOS-II.H

/*
 ** GLOBAL DEFINITIONS
 */

#define OSRTOS (ucos2cvu) /* Count of number of context switches */

#if (OS_VERSION >= 2)

#define OS_PDIR (ucos2cvl) /* Pointer to list of free task control blocks */
#define OS_PTAB (ucos2cv/ml) /* Table of entry control blocks */
#endif

#define OS_COUNT (ucos2cvr) /* Idle counter */

#define OS_CRT (ucos2cvz) /* Percentage of CPU used */
#define OS_MCB (ucos2cvm) /* Maximum value that Idle counter can take in 1 sec. */
#define OS_CCR (ucos2cvn) /* Value reached by idle counter at any time in 1 sec. */
#define OS_IOL (ucos2cvo) /* Flag indicating that the statistic task is ready */

#define OS_INTR (ucos2cvt) /* Interrupt nesting level */
#define OS_INTRU (ucos2cvu) /* Multitasking lock nesting level */
#define OS_INTRV (ucos2cvv) /* Priority of current task */
#define OS_INTRH (ucos2cvh) /* Priority of highest priority task */
#define OS_INTRM (ucos2cvm) /* Ready list group */

#define OS_MCB (ucos2cvo) /* Table of tasks which are ready to run */
#define OS_MCB (ucos2cvo) /* Flag indicating that kernel is running */

#define OS_TASKID (ucos2cvp) /* Number of tasks created */

#define OS_TASK (ucos2cvu) /* Pointer to currently running TCB */
#define OS_TASKO (ucos2cvu) /* Pointer to list of free TCBs */
#define OS_TASKN (ucos2cvu) /* Pointer to highest priority TCB ready to run */
#define OS_TASKM (ucos2cvu) /* Pointer to doubly linked list of TCBs */

#define OS_MCB (ucos2cvu) /* Table of pending to create TCBs */
#define OS_MCB (ucos2cvu) /* Current value of system time (in tick) */

extern TASKID osbnd(); /* Binds a task to a TCB */
extern reg regd(); /* Prepares the stack loading table */
extern reg regd(); /* Prepares the stack loading table */

{"summary":null}
Listing A.2 (continued)  uCOS_IT.B

/  
*/
/  FUNCTION MACROS
/  (Ignore Independent Functionality)
*/
/
/  /*************************************************/
/  MESSAGE QUEUE MANAGEMENT
/  
/  if (GQ_MEM
void OSPqueue(UI32 queue, U32 *event);   
void OSPsend(UI32 queue, U32 *event, UI64 *data);   
INT8U OSPfree(UI32 queue, UI32 *event);
MAILQ; 
ENDIF
/  /*************************************************/
/  HISTORY MANAGEMENT
/  
/  if (GQ_MEM
void OSPcheck(UI32 queue, UI32 *event, UI64 *data);
INT8U OSPquery(UI32 queue, UI32 *event);
ENDIF
/  /*************************************************/
/  MESSAGE QUEUE MANAGEMENT
/  
/  if (GQ_MEM
void OSPcheck(UI32 queue, UI32 *event);   
void OSPsend(UI32 queue, U32 *event, UI64 *data);   
INT8U OSPfree(UI32 queue, UI32 *event);
INT8U OSPquery(UI32 queue, UI32 *event);   
ENDIF
/  
/  /*MSG*/
/************* ENTERMEST MANAGEMENT *************/

/*

#if  OS_ESP_IN

INT16U OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/

#ifndef

OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/

/*

#ifndef

OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/

/*

#ifndef

OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/

/*

#ifndef

OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/

/*

#ifndef

OSKernelStart(void *start, void *pdata, OS_SUB_PROFILE);

#endif

*/
Listing A.2 (continued) \texttt{uCOS-II.H}

```c
/*
 * ***********************************************************************
 * \textbf{FORCE INTERRUPT}
 * ***********************************************************************
 */

void OSInit( void );

void  OSInitTick( void );

void  OSInitTick2( void );

void  OSInitTick3( void );

void  OSInitTick4( void );

void  NewKernel( void );

void  NewKernel2( void );

void  NewKernel3( void );

void  NewKernel4( void );

void  OSInit( void );

void  OSInitTick( void );

void  OSInitTick2( void );

void  OSInitTick3( void );

void  OSInitTick4( void );

/*SINCE*/
```
Listing A.2 (continued)  uCOS_IT.H

/**
 * INTERNAL FUNCTION PROTOTYPES
 * (Your uCOS/II Core MUST NOT call these functions)
 */

#ifndef _UCOS_IT_H
void OSStartCritical(void);
void OSUnlock(int32_t ID);
#endif

#ifndef _UCOS_MAX_THREADS
#define _UCOS_MAX_THREADS 2
#endif

#ifndef _UCOS_MAX_TASKS
#define _UCOS_MAX_TASKS 2
#endif

#ifndef _UCOS_MAX_SLOTS
#define _UCOS_MAX_SLOTS 2
#endif

#include "uc.h"
Listing A.2 (continued) uCOS_II.H

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource_h
Appendix B

Programming Conventions

Conventions should be established early in a project. These conventions are necessary to maintain co-
sistency throughout the project. Adopting conventions increases productivity and simplifies project
maintenance. A few years ago, I saw an article in the Hewlett-Packard Journal (see Bibliography on
page 585) about the processes used by a team of engineers to design the HP54720/10 oscilloscope. One
of the aspects of the design consisted of developing a coding convention. "A consistent format makes the
code much easier to read and understand. At the completion of the project, all of the engineers involved
were enthusiastic about using the standard in developing the code". If you are serious about improving
your programming skills you should get Code Complete by Steve McConnell (see Bibliography on
page 585). Steve also highly recommends that you adopt a coding convention before you begin pro-
gramming. As he says, "It's nearly impossible to change code to match your conventions after the code
is written".

In this section I will describe the conventions I have used to develop the software presented in this
book.

B.00 Directory Structure

Adopting a consistent directory structure avoids confusion when either more than one programmer is
involved in a project, or you are involved in many projects. This section shows the directory structure
that I use on a daily basis.

B.00.01 Directory Structure, Products

All software development projects are placed in a \PRODUCTS subdirectory from the root directory. I
prefer to create the \PRODUCTS subdirectory because it avoids having a large number of directories in
the root directory.

Each project is placed in a subdirectory by itself under the \PRODUCTS directory. Instead of having
all files in a project located in a single subdirectory, I like to split project related files in these subdiri-
tories. (There is nothing like looking at a project subdirectory containing dozens of files!). Each prod-
uct contains a number of subdirectories.

571
This subdirectory contains product specific software. It is assumed that you would use building blocks and thus this directory contains code that is actually specific to the product. The SOFTWARE directory further contains subdirectories such as:

- \PRODUCTS\project\SOFTWARE\SOURCE
  This subdirectory contains the actual product specific source code.
- \PRODUCTS\project\SOFTWARE\TEST
  This subdirectory contains the product build instructions (i.e., makefiles, scripts, batch files, etc.) to create a "test" version of the product to build.
- \PRODUCTS\project\SOFTWARE\OBJ
  This subdirectory contains the compiled and assembled code into relocatable object form of all the files needed to make the product.
- \PRODUCTS\project\SOFTWARE\VC
  This subdirectory contains the version controlled product specific software.
- \PRODUCTS\project\SOFTWARE\???
  You can have additional subdirectories that would contain documentation about the software aspects of your product (DOC directory), a directory where you could "collect" all the source files that make up your product in order to compile and assemble them (MAKE directory), a directory where you can "rebuild" any version of a released product (PROD directory), and more.

- \PRODUCTS\project\HARDWARE
  This subdirectory could contain information about the product's hardware (schematics, PCBs, parts lists, wire lists, etc.).
- \PRODUCTS\project\MECH
  This subdirectory could contain information about the mechanical aspects of your product (enclosures, injection molds, parts list, etc.).

### B.00.02 Directory Structure, Building Blocks

Each building block is placed in a subdirectory by itself under the SOFTWARE directory. The reason the building blocks are placed in a directory from the root is because the building blocks are not supposed to be platform dependent. Below each building block, I have the following subdirectories:

- SOFTWARE\building-block\SOURCE
  This subdirectory contains the source code of the building block.
- SOFTWARE\building-block \DOC
  This subdirectory contains documentation specific to the building block.
- SOFTWARE\building-block \VC
  The VC (Version Control) subdirectory contains version controlled archive files generated by a version control software package such as the Merant PVCS Version Manager (previously called PVCS). This subdirectory contains the revisions and versions of your source code, documentation, and executables. If you are new to version management and configuration building, consult the excellent book by Wayne A. Babich called Software Configuration Management or, contact Merant about their excellent software packages.

To remove the frustration of navigating through these subdirectories, I wrote a utility program that allows you to jump to a directory without having to use the DOS change directory command. This utility is called "\ aide" and is described in Appendix D.
B.01 C Programming Style

B.01.01 Overview

There are many ways to code a program in C (or any other language). The style you use is just as good as any other as long as you strive to attain the following goals:

- Portability
- Consistency
- Neatness
- Easy maintainance
- Easy understanding
- Simplicity

Whatever style you use, I would emphasize that it should be adopted consistently throughout all your projects. I would further insist that a single style be adopted by all team members on a large project. To this end, I would recommend that a C programming style document be formalized for your organization. Adopting a common coding style reduces code maintenance headaches and costs. Adopting a common style will avoid code rewrites. This section describes the C programming style I use.

The main emphasis on the programming style presented here is to make the source code easy to follow and maintain.

I don’t like to limit the width of my C source code to 80 characters just because today’s monitors only allow you to display 80 characters wide. My limitation is actually how many characters can be printed on an 8.5" × 11" page using compressed mode (17 characters per inch). Using compressed mode, you can accommodate up to 132 characters and have enough room on the left of the page for holes for insertion in a three ring binder. Allowing 132 characters per line prevents having uninteresting source code with comment lines. The code provided in this book uses 105 characters per line. This limitation is imposed by the publisher.

B.01.02 Header

The header of a C source file looks as shown below. Your company name and address can be on the first few lines followed by a title describing the contents of the file. A copyright notice is included to give warning of the proprietary nature of the software.
The name of the file is supplied followed by the name of the programmer(s). The name of the programmer who created the file is given first. The last item in the header is a description of the contents of the file.

I like to dictate when page breaks occur. This is done by inserting the special comment /*$PAGE*/ whenever you want a page break. The file is printed using a utility that I wrote called HPLISTC (see Appendix D). When HPLISTC encounters this comment, it sends a form feed character to the printer.

**0.01.03 Revision History**

Because of the dynamic nature of software, I always include a section in the source file to describe changes made to the file. You may either maintain version control manually or automate the process by using a version control software package. I prefer to use version control software because it takes care of a number of chores automatically. The revision control section contains the different revision levels, date and time and a short description of each of the different revision levels. Revision history should start on a page boundary.

**0.01.04 Include Files**

The header files needed for your project immediately follow the revision history section. You may either list only the header files required for the module or combine header files in a single header file.
Appendix 8: Programming Conventions — 575

like I do in a file called INCLUDES.H. I like to use an INCLUDES.H header file because it prevents you from having to remember which header file goes with which source file especially when new modules are added. The only inconvenience is that it takes longer to compile each file.

/*
 * INCLUDE FILES
 */

#include "INCLUDES.H"

/**/

B.01.05 Naming Identifiers

C compilers which conform to the ANSI X3J11 standard (most C compilers do by now) allow up to 32 characters for identifier names. Identifiers are variables, structure/union members, functions, macros, #defines, and so on. Descriptive identifiers can be formulated using this 32 character feature and the use of acronyms, abbreviations, and mnemonics (see the Acronym, Abbreviation, and Mnemonic Dictionary, Appendix C). Identifier names should reflect what the identifier is used for. I like to use a hierarchical method when creating an identifier. For instance, the function CSemaphore() indicates that it is part of the operating system (CS), it is a semaphore (Sem) and the operation being performed is to wait (Por5) for the semaphore. This method allows me to group all functions related to semaphores together.

Variable names should be declared on separate lines rather than combining them on a single line. Separate lines make it easy to provide a descriptive comment for each variable.

I use the filename as a prefix for variables that are either local (static) or global to the file. This makes it clear that the variables are being used locally and globally. For example, local and global variables of a file named KEY.C are declared as follows:

static INT16U KeyCharOut;    /* Number of keys pressed */
static char KeyStrmf[100];   /* Storage buffer to hold char */
char KeyStrCh;               /* Character typed */

/\SYNC/\n
Uppercase characters are used to separate words in an identifier. I prefer to use this technique versus making use of the underscore character, _ because underscores do not add any meaning to names and also use up character spaces.

Global variables (external to the file) can use any name as long as they contain a mixture of uppercase and lowercase characters and are prefixed with the module/file name (i.e., all global keyboard related variable names would be prefixed with the word KEY).

Formal arguments to a function and local variables within a function are declared in lowercase. The lowercase makes it obvious that such variables are local to a function. Global variables will contain a
mixture of upper- and lowercase characters. To make variables readable, you can use the underscore character (i.e., _).

Within functions, certain variable names can be reserved to always have the same meaning. Some examples are given below but others can be used as long as consistency is maintained.

\( i, j \) and \( x \)

for loop counters.

\( p_1, p_2 \ldots p_n \)

for pointers.

\( c, cl \ldots cn \)

for characters.

\( a, al \ldots an \)

for settings.

\( ia, iy \) and \( iz \)

for intermediate integer variables.

\( f_x, fy \) and \( fz \)

for intermediate floating point variables.

To summarize:

- **formal parameters** in a function declaration should only contain lowercase characters.
- **auto variable names** should only contain lowercase characters.
- **static variables and functions** should use the file/module name (or a portion of it) as a prefix and should make use of upper- and lowercase characters.
- **extern variables and functions** should use the file/module name (or a portion of it) as a prefix and should make use of upper- and lowercase characters.

### B.01.06 Acronyms, Abbreviations, & Mnemonics

When creating names for variables and functions (identifiers), it is often the practice to use acronyms (e.g., OS, ISR, PCI and so on), abbreviations (Buf, doc, etc.) and mnemonics (clr, cmp, etc.). The use of acronyms, abbreviations, and mnemonics allows an identifier to be descriptive while reserving fewer characters. Unfortunately, if acronyms, abbreviations, and mnemonics are not used consistently, they may add confusion. To ensure consistency, I have opted to create a list of acronyms, abbreviations, and mnemonics that I use in all my projects. The same acronym, abbreviation, or mnemonic is used throughout, once it is assigned. I call this list the Acronym, Abbreviation, and Mnemonic Dictionary (see Appendix C). As I need more acronyms, abbreviations, or mnemonics, I simply add them to the list.

There might be instances where one list for all products doesn’t make sense. For instance, if you are an engineering firm working on a project for different clients and the products that you develop are totally unrelated, then a different list for each project would be more appropriate; the vocabulary for the farming industry is not the same as the vocabulary for the defense industry. I use the rule that if all products are similar, they use the same dictionary.

A common dictionary to a project team will also increase the team’s productivity. It is important this consistency be maintained throughout a project, irrespective of the individual programmer(s). Once buf has been agreed to mean “buffer” it should be used by all project members instead of having some individuals use buffer and others use bfr. To further this concept, you should always use buf even if your identifier can accommodate the full name; stick to buf even if you can fully write the word “buffer.”

Appendix C provides the acronyms, abbreviations, and mnemonics dictionary that I used for this book. Note that some of the words are the same in both columns. This is done to indicate that there is no acronym, abbreviation, or mnemonic which would better describe the word on the left.
B.01.07 Comments

I find it very difficult to mentally separate code from comments when code and comments are intertwined. Because of this, I never write code with comments. Comments are written to the right of the actual C code. When large comments are necessary, they are written in the function description header.

Comments are lined up as shown in the following example. The comment terminators (//) do not need to be lined up, but for neatness I prefer to do so. It is not necessary to have one comment per line since a comment could apply to a few lines.

```c
/*
 * Description: Function to convert string 's' to an integer.
 * Arguments: ASCII string to convert to integer.
 * Returns: String converted to an 'int'.
 */

int strn_to_int(const char *s)
{
    int n; /* Partial result of conversion */
    n = 0; /* Initialize result */
    while (*s >= '0' && *s <= '9')
    { /* for all valid characters and end of string */
        *s--; /* Convert char to int and add to partial result */
    }
    return n; /* Return the result of the converted string */
}

/* other */
```

B.01.08 Defines

Header files (.h) and C source files (.c) might require that constants and macros be defined. Constants and macros are always written in uppercase with the underscore character used to separate words. Note that hexadecimal numbers are always written with a lowercase x and all uppercase letters for hexadecimal A through F.
578 — Embedded Systems Building Blocks, Second Edition

/**
 * @startuml
 * constants & macros
 *
 */
#define KEY_FF 0x00
#define KEY_CR 0x0D
#define KEY_KEYFULL() (KeyNum > 0)
/**$\textbf{BEGIN}$*/

**B.01.09 Data Types**

C allows you to create new data types using the <tt>typedef</tt> keyword. I declare all data types using upper-case characters, and thus follow the same rule used for constants and macros. There is never a problem confusing constants, macros, and data types because of the context in which they are used. Since different microprocessors have different word lengths, I like to declare the following data types (assuming Borland C++ V4.5):

```c
/**
 * @startuml
 * data types
 *
 */
typedef unsigned char BOOLEANS; /* Boolean */
typedef unsigned char I8BIT; /* 8 bit unsigned */
typedef unsigned int I16BIT; /* 16 bit unsigned */
typedef int I16BITS; /* 16 bit signed */
typedef unsigned long I32BIT; /* 32 bit unsigned */
typedef long I32BITS; /* 32 bit signed */
typedef float FP; /* Floating Point */
/**$\textbf{END}$*/
```

Using these <tt>typedefs</tt>, you will always know the size of each data type.

**B.01.010 Local Variables**

Some source modules will require that local variables be available. These variables are only needed for the source file (file scope) and should thus be hidden from the other modules. Hiding these variables is
accomplished in C by using the static keyword. Variables can either be listed in alphabetical order or in functional order.

/*
*   ***********************************************************************
*   LOCAL VARIABLES
*   ***********************************************************************
*/
static char KeyBuf[100];
static INT16U KeyNRd;

"/\$PAGE*/\

B.01.01 Function Prototypes
This section contains the prototypes (or calling conventions) used by the functions declared in the file. The order in which functions are prototyped should be the order in which the functions are declared in the file. This order allows you to quickly locate the position of a function when the file is printed.

/*
*   ***********************************************************************
*   FUNCTION PROTOTYPES
*   ***********************************************************************
*/
void4 KeyClrBuf(void);
static BOOL8F KeyChkStat(void);
static INT16U KeyGetCtrl(int clr);

"/\$PAGE*/\

Also note that the static keyword, the returned data type, and the function names are all aligned.

B.01.02 Function Declarations
As much as possible, there should only be one function per page when code listings are printed on a printer. A comment block should precede each function. All comment blocks should look as shown below. A description of the function should be given and should include as much information as necessary. If the combination of the comment block and the source code extends past a printed page, a page break should be forced (preferably between the end of the comment block and the start of the function). This allows the function to be on a page by itself and prevents having a page break in the middle of the function. If the function itself is longer than a printed page then it should be broken by a page break comment ("/\$PAGE*/") in a logical location (i.e., at the end of an if statement instead of in the middle of one).
More than one small function can be declared on a single page. They should all, however, contain the comment block describing the function. The beginning of a function should start at least two lines after the end of the previous function.

```c
/*
 * DESCRIPTION: Flush keyboard buffer
 * ARGUMENTS: none
 * RETURNS: none
 * NOTES: none
 */

void KeyClrBuf() {
}
```

Functions that are only used within the file should be declared static to hide them from other functions in different files.

By convention, I always call all invocations of the function without a space between the function name and the open parenthesis of the argument list. Because of this, I place a space between the name of the function and the opening parenthesis of the argument list in the function declaration as shown above. This is done so that I can quickly find the function definition using a grep utility.

Function names should make use of the filename as a prefix. This prefix makes it easy to locate function declarations in medium to large projects. It also makes it very easy to know where these functions are declared. For example, all functions in a file named KEY.C and functions in a file named VIDEO.C could be declared as follows:

- **KEY.C**
  - KeyGetChar()
  - KeyGetLine()
  - KeyGetPutKey()

- **VIDEO.C**
  - VideoGetAttr()
  - VideoPutChar()
  - VideoPutStr()
  - VideoGetAttr()

It’s not necessary to use the whole filename as a prefix. For example, a file called KEYBOARD.C could have functions starting with Key instead of Keyboard. It is also preferable to use uppercase characters to separate words in a function name instead of using underscores. Again, underscores don’t add any meaning to names and they use up character spaces. As mentioned previously, formal parameters and local variables should be in lowercase. This makes it clear that such variables have a scope limited to the function.
Each local variable name must be declared on its own line. This allows the programmer to comment each one as needed. Local variables are indented four spaces. The statements for the function are separated from the local variables by three spaces. Declarations of local variables should be physically separated from the statements because they are different.

B.01.05 Indentation

Indentation is important to show the flow of the function. The question is, how many spaces are needed for indentation? One space is obviously not enough while 8 spaces is too much. The compromise I use is four spaces. I also never use TABs, because various printers will interpret TABs differently; and your code may not look as you want. Avoiding TABs does not mean that you can’t use the TAB key on your keyboard. A good editor will give you the option to replace TABs with spaces (in this case, 4 spaces).

A space follows the keywords if, for, while, and do. The keyword else has the privilege of having one before and one after it if curly braces are used. I write if (condition) on its own line and the statement(s) to execute on the next following line(s) as follows:

```java
if (x < 0)
    x = 23;

if (y > 20 { 
  x = 10;
  y = 100;
  p++;
});
```

instead of the following method:

```java
if (x < 0) { 
    x = 23;
}
if (y > 20) (x = 10; x = 100; p++);
```

There are two reasons for this method. The first is that I like to keep the decision portion apart from the execution statement(s). The second reason is consistency with the method I use for while, for, and do statements.

```java
switch statements are treated in any other conditional statement. Note that the case statements are lined up with the case label. The important point here is that switch statements must be easy to follow. Cases should also be separated from one another.

if (x = 0) 
    y = 20;
    z = 1;
```


```c
if (z < LDM) {
    x = y + z;
    z = 10;
} else {
    x = y - z;
    z = -25;
}

for (i = 0; i < MAX_ITER; i++) {
    *p2++ = *p1++;
    mx[i] = 0;
}

while (*p1) {
    *p2++ = *p3++;
    cnt++;
}

switch (key) {
    case KEY_BS:
        if (cnt > 0) {
            p--; 
            cnt--;
        }
        break;
    case KEY_CR:
        *p = *x;
        break;
    case KEY_LINES_FEED:
        p++;
        break;
    default:
        *p++ = key;
        cnt++;
        break;
}
```
do {
    if (not -f
        *p2++ = *p1++;
    } while (cnt > 0);

8.01.04 Statements & Expressions

All statements and expressions should be made to fit on a single source line. I never use more than one
assignment per line such as:

    x = y = z = 1;

Even though this is correct in C, when the variable names get more complicated, the intent might not
be as obvious.

The following operators are written with no space around them:

- Structure pointer operator
  p->m
- Structure member operator
  s.m
- Array subscripting
  n[i]

Parentheses after function names have no space(s) before them. A space should be introduced after
each comma to separate each actual argument in a function. Expressions within parentheses are written
with no space after the opening parenthesis and no space before the closing parenthesis. Commas and
semicolons should have one space after them.

    stmt(t, s, n);
    for (i = 0; i < m; i++)

The unary operators are written with no space between them and their operands:

    !p  -b  ++i  --j (length)  *p  &x  size_t(x)

The binary operators is precued and followed by one or more spaces, as is the ternary operator:

    c1 < c2  x + y  i += 2  n > 0 ? n : -n

The keywords if, while, for, match, and return are followed by one space.

For assignments, numbers are lined up in columns as if you were to add them (assuming you hard-
code numbers). The equal signs are also lined up.

    x = 100.567;
    temp = 12.700;
    var3 = 0.768;
    variable += 12;
    storage = &var[0];
B.01.015 Structures and Unions

Structures are typedefd since this allows a single name to represent the structure. The structure type is declared using all uppercase characters with underscore characters used to separate words.

```
typedef struct line { /* Structure that defines a LINE */
  int LineStartX; /* 'X' & 'Y' starting coordinate */
  int LineStartY; /* 'X' & 'Y' ending coordinate */
  int LineEndX;
  int LineEndY;
  int LineColor; /* Color of line to draw */
} LINE;
```

```
typedef struct point { /* Structure that defines a POINT */
  int PointX; /* 'X' & 'Y' coordinate of point */
  int PointY;
  int PointColor; /* Color of point */
} POINT;
```

Structure members start with the same prefix (as shown in the examples above). Member names should start with the name of the structure type (or a portion of it). This makes it clear when pointers are used to reference members of a structure such as:

```
p-LineColor; /* We know that 'p' is a pointer to LINE */
```

B.01.016 Reserved Keywords

The following keywords should never be used for identifiers. These keywords are reserved in the C++ language as defined by Bjarne Stroustrup and are thus reserved for future compatibility.

- asm
- class
- delete
- overload
- private
- protected
- public
- friend
- handle
- new
- operator
- template
- this
- virtual
B.02 Bibliography

Bibich, Wayne A.
Software Configuration Management
Reading, Massachusetts
Addison-Wesley Publishing Company, 1986
ISBN 0-201-10161-1

Long, David W. and Deff, Christopher P.
A Survey of Processes Used in the Development of Firmware for a
Multiprocessor Embedded System
Hewlett-Packard Journal, October 1993, p.59-65

McConnell, Steve
Code Complete
Redmond, Washington
Microsoft Press, 1993
ISBN 1-55615-484-4

Merant, Inc.
PVS 3 Version Manager
735 SW 158th Avenue
Beaverton, OR 97006
(503) 645-1150

Merant, Inc.
PVS 3 Configuration Builder
735 SW 158th Avenue
Beaverton, OR 97006
(503) 645-1150
Acronym, Abbreviation, and Mnemonic Dictionary

Naming functions and variables might seem trivial but good function and variable names are a sign of superior programs. When creating names for variables and functions (identifiers), it is often the practice to use acronyms (e.g., Q5, IIR, TCB), abbreviations (buf, doc, etc.), and mnemonics (clr, crp, etc.). The use of acronyms, abbreviations, and mnemonics allows an identifier to be descriptive while requiring fewer characters. Unfortunately, if acronyms, abbreviations, and mnemonics are not used consistently, they may add confusion. To ensure consistency, I created a list of acronyms, abbreviations, and mnemonics that I use in all my projects. Once assigned the same acronym, abbreviation, or mnemonic is used consistently. I call this list the Acronym, Abbreviation, and Mnemonic Dictionary. As I need more acronyms, abbreviations, or mnemonics, I simply add them to the list.

Table C.1 shows the acronyms, abbreviations, and mnemonics dictionary that I used for this book. Note that some of the words are the same in both columns. This is done to indicate that there is no acronym, abbreviation, or mnemonic which would better describe the word on the left. A shaded entry in Table C.1 indicates that an acronym, abbreviation, or mnemonic has been used.

You can combine acronyms, abbreviations, and mnemonics to make up a full function or variable name. For example:

1. Calculate Cursor Position could be CurCalcPos.
2. Get Keyboard Buffer could be KeybufGet.
3. Clear Counter Group could be ClrCntGrp.
4. Clear Alarm Status could be NClearAlm.

In fact, I prefer to group related items by their names. You may have noticed that all functions and variables within each module in this book start with the acronym, abbreviation, or mnemonic of the module (or file) name. This allows you to quickly know where each function or variable is declared.
# Table C.1 Acronyms, abbreviations, and mnemonics dictionary

<table>
<thead>
<tr>
<th>Description</th>
<th>Acronym, abbreviation, or mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Addition</td>
<td>Add</td>
</tr>
<tr>
<td>2 Action</td>
<td>Act</td>
</tr>
<tr>
<td>3 Analog Input(s)</td>
<td>AI</td>
</tr>
<tr>
<td>4 Analog I/O</td>
<td>Ali</td>
</tr>
<tr>
<td>5 All</td>
<td>All</td>
</tr>
<tr>
<td>6 Alarm</td>
<td>Alm</td>
</tr>
<tr>
<td>7 Analog Output(s)</td>
<td>AO</td>
</tr>
<tr>
<td>8 Argument(s)</td>
<td>Arg</td>
</tr>
<tr>
<td>9 Bar</td>
<td>Bar</td>
</tr>
<tr>
<td>10 Bit</td>
<td>Bit</td>
</tr>
<tr>
<td>11 Buffer</td>
<td>Buf</td>
</tr>
<tr>
<td>12 Bypass</td>
<td>Bypass</td>
</tr>
<tr>
<td>13 Calibration</td>
<td>Cal</td>
</tr>
<tr>
<td>14 Calculate</td>
<td>Calc</td>
</tr>
<tr>
<td>15 Configuration</td>
<td>Cfg</td>
</tr>
<tr>
<td>16 Channel</td>
<td>Ch</td>
</tr>
<tr>
<td>17 Change</td>
<td>Change</td>
</tr>
<tr>
<td>18 Check</td>
<td>_chk</td>
</tr>
<tr>
<td>19 Clock</td>
<td>Clk</td>
</tr>
<tr>
<td>20 Clear</td>
<td>Ch</td>
</tr>
<tr>
<td>21 Clear Screen</td>
<td>Chs</td>
</tr>
<tr>
<td>22 Command</td>
<td>Cmd</td>
</tr>
<tr>
<td>23 Compare</td>
<td>Cmp</td>
</tr>
<tr>
<td>24 Count</td>
<td>Cnt</td>
</tr>
<tr>
<td>25 Column</td>
<td>Col</td>
</tr>
<tr>
<td>26 Communication</td>
<td>Comm</td>
</tr>
<tr>
<td>27 Control</td>
<td>Ctrl</td>
</tr>
<tr>
<td>28 Context</td>
<td>Ctx</td>
</tr>
<tr>
<td>29 Current</td>
<td>Curr</td>
</tr>
<tr>
<td>30 Cursor</td>
<td>Curser</td>
</tr>
<tr>
<td>31 Control Word</td>
<td>CW</td>
</tr>
<tr>
<td>32 Date</td>
<td>Date</td>
</tr>
<tr>
<td>33 Day</td>
<td>Day</td>
</tr>
<tr>
<td>34 Debounce</td>
<td>Debounce</td>
</tr>
<tr>
<td>35 Decimal</td>
<td>Dec</td>
</tr>
<tr>
<td>Description</td>
<td>Acronym, abbreviation, or mnemonic</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Decoded</td>
<td>Decode</td>
</tr>
<tr>
<td>Define</td>
<td>Def</td>
</tr>
<tr>
<td>Delete</td>
<td>Del</td>
</tr>
<tr>
<td>Detect/Detection</td>
<td>Detect</td>
</tr>
<tr>
<td>Discrete Input(s)</td>
<td>Di</td>
</tr>
<tr>
<td>Digit</td>
<td>Dig</td>
</tr>
<tr>
<td>Discrete I/O</td>
<td>DIO</td>
</tr>
<tr>
<td>Disable</td>
<td>Dis</td>
</tr>
<tr>
<td>Display</td>
<td>Disp</td>
</tr>
<tr>
<td>Division.</td>
<td>Div</td>
</tr>
<tr>
<td>Divisor</td>
<td>Dr</td>
</tr>
<tr>
<td>Division</td>
<td>Div</td>
</tr>
<tr>
<td>Delay</td>
<td>Dly</td>
</tr>
<tr>
<td>Discrete Output(s)</td>
<td>DO</td>
</tr>
<tr>
<td>Day-of-week</td>
<td>DOW</td>
</tr>
<tr>
<td>Down</td>
<td>Down</td>
</tr>
<tr>
<td>Dummy</td>
<td>Dummy</td>
</tr>
<tr>
<td>Edge</td>
<td>Edge</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty</td>
</tr>
<tr>
<td>Enable</td>
<td>En</td>
</tr>
<tr>
<td>Enter</td>
<td>Enter</td>
</tr>
<tr>
<td>Entries</td>
<td>Entries</td>
</tr>
<tr>
<td>Error(s)</td>
<td>Err</td>
</tr>
<tr>
<td>Engineering 'Units'</td>
<td>EU</td>
</tr>
<tr>
<td>Event(s)</td>
<td>Event</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit</td>
</tr>
<tr>
<td>Exposure</td>
<td>Exp</td>
</tr>
<tr>
<td>Flag</td>
<td>Flg</td>
</tr>
<tr>
<td>Flash</td>
<td>Flsh</td>
</tr>
<tr>
<td>Function(s)</td>
<td>Fct</td>
</tr>
<tr>
<td>Format</td>
<td>Format</td>
</tr>
<tr>
<td>Fraction</td>
<td>Frc</td>
</tr>
<tr>
<td>Free</td>
<td>Free</td>
</tr>
<tr>
<td>Full</td>
<td>Fll</td>
</tr>
<tr>
<td>Gain</td>
<td>Gain</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>---------------------</td>
</tr>
<tr>
<td>71</td>
<td>Get</td>
</tr>
<tr>
<td>72</td>
<td>Group(s)</td>
</tr>
<tr>
<td>73</td>
<td>Handler</td>
</tr>
<tr>
<td>74</td>
<td>Hexadecimal</td>
</tr>
<tr>
<td>75</td>
<td>High</td>
</tr>
<tr>
<td>76</td>
<td>Hit</td>
</tr>
<tr>
<td>77</td>
<td>High Priority Task</td>
</tr>
<tr>
<td>78</td>
<td>Hour(s)</td>
</tr>
<tr>
<td>79</td>
<td>I.D.</td>
</tr>
<tr>
<td>80</td>
<td>Idle</td>
</tr>
<tr>
<td>81</td>
<td>Input(s)</td>
</tr>
<tr>
<td>82</td>
<td>Initialization</td>
</tr>
<tr>
<td>83</td>
<td>Initialize</td>
</tr>
<tr>
<td>84</td>
<td>Interrupt</td>
</tr>
<tr>
<td>85</td>
<td>Invert</td>
</tr>
<tr>
<td>86</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>87</td>
<td>Index</td>
</tr>
<tr>
<td>88</td>
<td>Key</td>
</tr>
<tr>
<td>89</td>
<td>Keyboard</td>
</tr>
<tr>
<td>90</td>
<td>Limit</td>
</tr>
<tr>
<td>91</td>
<td>List</td>
</tr>
<tr>
<td>92</td>
<td>Low</td>
</tr>
<tr>
<td>93</td>
<td>Lower</td>
</tr>
<tr>
<td>94</td>
<td>Lowest</td>
</tr>
<tr>
<td>95</td>
<td>Lock</td>
</tr>
<tr>
<td>96</td>
<td>Low Priority Task</td>
</tr>
<tr>
<td>97</td>
<td>Mantissa</td>
</tr>
<tr>
<td>98</td>
<td>Manual</td>
</tr>
<tr>
<td>99</td>
<td>Maximum</td>
</tr>
<tr>
<td>100</td>
<td>Mailbox</td>
</tr>
<tr>
<td>101</td>
<td>Minimum</td>
</tr>
<tr>
<td>102</td>
<td>Minutes(s)</td>
</tr>
<tr>
<td>103</td>
<td>Mode</td>
</tr>
<tr>
<td>104</td>
<td>Month</td>
</tr>
<tr>
<td>105</td>
<td>Message</td>
</tr>
<tr>
<td>Description</td>
<td>Acronym, abbreviation, or mnemonic</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Mask</td>
<td>Msk</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Mlt</td>
</tr>
<tr>
<td>Multiplex</td>
<td>Mux</td>
</tr>
<tr>
<td>Number of</td>
<td>N</td>
</tr>
<tr>
<td>Nesting</td>
<td>Nesting</td>
</tr>
<tr>
<td>New</td>
<td>New</td>
</tr>
<tr>
<td>Next</td>
<td>Next</td>
</tr>
<tr>
<td>Offset</td>
<td>Offset</td>
</tr>
<tr>
<td>cfile</td>
<td>Old</td>
</tr>
<tr>
<td>Operating System</td>
<td>OS</td>
</tr>
<tr>
<td>Output</td>
<td>Out</td>
</tr>
<tr>
<td>Overflow</td>
<td>Ovf</td>
</tr>
<tr>
<td>Pass</td>
<td>Pass</td>
</tr>
<tr>
<td>Port</td>
<td>Port</td>
</tr>
<tr>
<td>Position</td>
<td>Pos</td>
</tr>
<tr>
<td>Previous</td>
<td>Prev</td>
</tr>
<tr>
<td>Priority</td>
<td>Prio</td>
</tr>
<tr>
<td>Printer</td>
<td>Prt</td>
</tr>
<tr>
<td>Pointer</td>
<td>Prt</td>
</tr>
<tr>
<td>Put</td>
<td>Put</td>
</tr>
<tr>
<td>Queue</td>
<td>Q</td>
</tr>
<tr>
<td>Raw</td>
<td>Raw</td>
</tr>
<tr>
<td>'Recall</td>
<td>Rd</td>
</tr>
<tr>
<td>Read</td>
<td>Rd</td>
</tr>
<tr>
<td>Ready</td>
<td>Bly</td>
</tr>
<tr>
<td>Register</td>
<td>Reg</td>
</tr>
<tr>
<td>Reset</td>
<td>Rest</td>
</tr>
<tr>
<td>Resume</td>
<td>Resume</td>
</tr>
<tr>
<td>Ring</td>
<td>Ring</td>
</tr>
<tr>
<td>Row</td>
<td>Row</td>
</tr>
<tr>
<td>Repeat</td>
<td>Rpe</td>
</tr>
<tr>
<td>Real-Time</td>
<td>RT</td>
</tr>
<tr>
<td>Running</td>
<td>Running</td>
</tr>
<tr>
<td>Receive</td>
<td>Rx</td>
</tr>
<tr>
<td>Scale</td>
<td>Scale</td>
</tr>
<tr>
<td>Description</td>
<td>Acronym, abbreviation, or mnemonic</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>141 Scaler</td>
<td>Scaling</td>
</tr>
<tr>
<td>142 Scan</td>
<td>Scan</td>
</tr>
<tr>
<td>143 Schedule</td>
<td>Sched</td>
</tr>
<tr>
<td>144 Scheduler</td>
<td>Sched</td>
</tr>
<tr>
<td>145 Screen</td>
<td>Scr</td>
</tr>
<tr>
<td>146 Second(s)</td>
<td>Sec</td>
</tr>
<tr>
<td>147 Segment(s)</td>
<td>Seg</td>
</tr>
<tr>
<td>148 Select</td>
<td>Sel</td>
</tr>
<tr>
<td>149 Semaphore</td>
<td>Sem</td>
</tr>
<tr>
<td>150 Set</td>
<td>Set</td>
</tr>
<tr>
<td>151 Scale Factor</td>
<td>SF</td>
</tr>
<tr>
<td>152 Size</td>
<td>Size</td>
</tr>
<tr>
<td>153 Seven-segments</td>
<td>SS</td>
</tr>
<tr>
<td>154 Start</td>
<td>Start</td>
</tr>
<tr>
<td>155 Statistic(s)</td>
<td>Stat</td>
</tr>
<tr>
<td>156 Status</td>
<td>Stat</td>
</tr>
<tr>
<td>157 State</td>
<td>Staue</td>
</tr>
<tr>
<td>158 Stack</td>
<td>Stk</td>
</tr>
<tr>
<td>159 Stop</td>
<td>Stop</td>
</tr>
<tr>
<td>160 String</td>
<td>Sr</td>
</tr>
<tr>
<td>161 Subtraction</td>
<td>Sub</td>
</tr>
<tr>
<td>162 Suspend</td>
<td>Suspended</td>
</tr>
<tr>
<td>163 Switch</td>
<td>Sw</td>
</tr>
<tr>
<td>164 Synchronize</td>
<td>Sync</td>
</tr>
<tr>
<td>165 Task</td>
<td>Task</td>
</tr>
<tr>
<td>166 Table</td>
<td>Tbl</td>
</tr>
<tr>
<td>167 Threshold</td>
<td>Th</td>
</tr>
<tr>
<td>168 Tick</td>
<td>Tick</td>
</tr>
<tr>
<td>169 Time</td>
<td>Time</td>
</tr>
<tr>
<td>170 Timer</td>
<td>Tme</td>
</tr>
<tr>
<td>171 Trigger</td>
<td>Trig</td>
</tr>
<tr>
<td>172 Time-stamp</td>
<td>TS</td>
</tr>
<tr>
<td>173 Transmit</td>
<td>Tx</td>
</tr>
<tr>
<td>174 Unlock</td>
<td>Unlock</td>
</tr>
<tr>
<td>175 Up</td>
<td>Up</td>
</tr>
<tr>
<td>Description</td>
<td>Acronym, abbreviation, or mnemonic</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Update</td>
<td>Update</td>
</tr>
<tr>
<td>Value</td>
<td>Val</td>
</tr>
<tr>
<td>Vector</td>
<td>Vect</td>
</tr>
<tr>
<td>Write</td>
<td>Wr</td>
</tr>
<tr>
<td>Year</td>
<td>Year</td>
</tr>
</tbody>
</table>

Table C.1  Acronyms, abbreviations, and mnemonics dictionary
Appendix D

HPLISTC and TO

HPLISTC and TO are MS-DOS utilities that are provided in both executeable and source form for your convenience.

D.00 HPLISTC

HPLISTC is an MS-DOS utility to print C source files on an HP Laserjet printer. HPLISTC will print your source code in compressed mode; 17 characters per inch (CPI). An 8 1/2" x 11" page (portrait) will accommodate up to 132 characters. An 11" x 8 1/2" page (landscape) will accommodate up to 175 characters. Once the source code is printed, HPLISTC returns the printer to its normal print mode.

The main directory for HPLISTC is C:\SOFTWARE\HPLISTC. HPLISTC is provided in two files:
HPLISTC.EXE (see C:\SOFTWARE\HPLISTC\EXE) is the MS-DOS executable and HPLISTC.C (see C:\SOFTWARE\HPLISTC\SOURCE) is the source code.

HPLISTC prints the current date and time, the filename, its extension, and the page number at the top of each page. An optional title can also be printed at the top of each page. As HPLISTC prints the source code, it looks for two special comments: /*TITLE=*/ or /*Title=*/ and /*SPACE*/ or /*space*/.

The /*TITLE=*/ comment is used to specify the title to be printed on the second line of each page. You can define a new title for each page by using the /*TITLE=*/ comment. The new title will be printed at the top of the next page. For example:

/*TITLE=Matrix Keyboard Driver*/

will set the title for the next page to Matrix Keyboard Driver, and this title will be printed on each subsequent page of your source code until the title is changed again.

The /*SPACE*/ comment is used to force a page break in your source code listing. HPLISTC will not eject the page unless you specifically specify the /*SPACE*/ comment. If you do not force a page break using the /*SPACE*/ comment, a short function may be printed on two separate pages if a page break is forced by the printer when it reaches its maximum number of lines per page. The page number

595
on the top of each page actually indicates the number of occurrences of the \*PAGE\* comment encountered by LISTC or REPLISTC.

Before each line is printed, REPLISTC prints a line count that can be used for reference purposes. REPLISTC also allows you to print source code in landscape mode. The programs are invoked as follows:

```
REPLISTC filename.ext [L l | I i | D d] [destination]
```

where filename.ext is the name of the file to print and destination is the destination of the printout. Since REPLISTC sends the output to stdout, the printout can be redirected to a file, a printer (LPT1, LPT2, etc.), or a COM port (COM1, COM2, etc.) by using the MS-DOS redirector >. By default, REPLISTC outputs to the monitor.

L or l (lowercase l) means to print the file in landscape mode, allowing you to print about 175 columns wide.

**D.01 TO**

TO is an MS-DOS utility that allows you to go to a directory without having to type:

```
CD path
```

or

```
CD ..\path
```

TO is probably the MS-DOS utility I use the most because it allows me to move between directories very quickly. At the DOS prompt, you simply type TO followed by the name you associated with a directory and then press Enter as follows:

```
TO name
```

where name is a name you associated with a path. The names and paths are placed in an ASCII file called TO.TBL, which resides in the root directory of the current drive. TO scans TO.TBL for the name you specified on the command line. If the name exists in TO.TBL, the directory is changed to the path specified with the name. If name is not found in TO.TBL, the message Invalid NAME. is displayed.

The main directory for TO is C: \SOFTWARE\TO. TO is provided in three files: TO.EXE (see C: \SOFTWARE\TO \USER) is the MS-DOS executable; TO.TBL is an example of the correspondence table between your name and the desired directory associated with this name (see C: \SOFTWARE\TO \DEMO), and TO.C (see C: \SOFTWARE\TO \SOURCE) is the source code.

The format of TO.TBL is shown in Listing D.1. Note that the name must be separated from the desired path by a comma.
Listing D.1  Format of TO.TBL

name, path
name, path
... ...
... ...
name, path

An example of TO.TBL is shown in Listing D.2.

Listing D.2  Example of TO.TBL

A, \SOURCE
C, \SOURCE
D, \DOC
L, \LST
O, \OBJ
P, \PRO
T, \TEST
W, \WORK
XO., \SOFTWARE \BLOCKS \ALTO \SOURCE
CLM, \SOFTWARE \BLOCKS \CLI \SOURCE
COMM, \SOFTWARE \BLOCKS \COMM \SOURCE
DID, \SOFTWARE \BLOCKS \DID \SOURCE
DDBL-PP, \SOFTWARE \VOCES-II \DDBL-PP
KEY_MK, \SOFTWARE \BLOCKS \KEY_MK \SOURCE
LCD, \SOFTWARE \BLOCKS \LCD \SOURCE
LED, \SOFTWARE \BLOCKS \LED \SOURCE
LISTC, \SOFTWARE \BLOCKS \LISTC \SOURCE
NLG, \SOFTWARE \BLOCKS \NLG \SOURCE
TO, \SOFTWARE \TO \SOURCE
VCSL, \SOFTWARE \VCSL \SOURCE
VCSL-II, \SOFTWARE \VCSL-II \SOURCE

You may optionally add an envy by typing the path associated with a name on the command line prompt as follows:

TO name path
In this case, `TO` will append this new entry at the end of `TO`. This avoids having to use a text editor to add a new entry to `TO`. Similarly, if you type:

```
TO ALQ
```

then `TO` will change directory to `\SOFTMEM\BLOCKS\ALQ\SOURCE`. If you type:

```
TO CLK
```

then `TO` will change directory to `\SOFTMEM\BLOCKS\CLK\SOURCE`. `TO` can be as long as needed, but each name must be unique. Note that two names can be associated with the same directory. If you add entries in `TO`, `TEL` using a text editor, all entries must be entered in uppercase. When you invoke `TO` at the DOS prompt, the name you specify is converted to uppercase before the program searches through the table. `TO` looks for the first entry that matches the directory name in the file. For faster response, you may want to place your most frequently used directories at the beginning of the file.
Appendix E

Companion CD-ROM

R&D Books has included a companion CD-ROM to *Embedded Systems Building Blocks, Complete and Ready-to-Use Modules in C*. The CD-ROM is in MS-DOS format and contains all the source code provided in this book. The data sheets of the electronic components I have used are also on the companion CD-ROM in PDF format.

### E.00 Hardware/Software Requirements

<table>
<thead>
<tr>
<th>Equipment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>PC/AT compatible system</td>
</tr>
<tr>
<td>Fixed Disk Capacity</td>
<td>5 Megabytes free</td>
</tr>
<tr>
<td>System Memory</td>
<td>640K bytes of RAM</td>
</tr>
<tr>
<td>Operating System</td>
<td>MS-DOS, Windows 95, Windows 98, or Windows NT</td>
</tr>
</tbody>
</table>

### E.01 Installation

Use the `Install.bat` file to decompress and transfer the ESBB files from the CD to your system. The `Install.bat` expects 2 arguments:

1. Load DOS or open a DOS window under Windows 95/98/NT and specify the C: drive as the default drive.
2. Insert the CD-ROM in your CD drive.
3. Type `<c:drive>:INSTALL <cd-drive> [destination]` where `<c:drive>` is the drive letter of your CD-ROM and `[destination]` is the drive letter where you want ESBB installed. For example, to install ESBB on your hard disk drive E: from a CD drive H:, you would type:

   ```
   H: INSTALL H E
   ```
E.02 Directory Structure

Once INSTALL has completed, your destination drive will contain the following subdirectories:

- **SOFTWARE**

  The main directory from where all software-related files are placed.

- **SOFTWARE\BLOCKS**

  The main directory where all building blocks are located.

- **SOFTWARE\BLOCKS\AI0\SOURCE**

  This directory contains the source code for the analog I/O module (Chapter 10). The files in this directory are AI0.C and AI0.H.

- **SOFTWARE\BLOCKS\COM\SOURCE**

  This directory contains the source code for the asynchronous serial communication modules COM1, COM2, COM3, and COM4 (Chapter 11). The files in this directory are:
  - COM2.C and COM2.H
  - COM3.C and COM3.H

- **SOFTWARE\BLOCKS\KEY\SOURCE**

  This directory contains the source code for the keyboard scanning module presented in Chapter 3. The source files are KEY.C and KEY.H.

- **SOFTWARE\BLOCKS\LCD\SOURCE**

  This directory contains the source code for the character LCD module presented in Chapter 5. The source files are LCD.C and LCD.H.

- **SOFTWARE\BLOCKS\LED\SOURCE**

  This directory contains the source code for the multiplexed LED module presented in Chapter 4. The source files are LED.C, LED.D, LED.R, and LED.H.

NOTE: Make sure you read the README file on the companion CD-ROM for last minute changes and notes.
Appendix E: Companion CD-ROM — 601

- \\
  \$SOFTWARE\BLOCKS\PC\HDL45
  This directory contains the source code for PC related services (see Chapter 1). The files in this
directory are PC.C and PC.H.

- \$SOFTWARE\BLOCKS\SAMPLE\SOURCE
  This directory contains the source code for the sample code (see Chapter 1). The files in this direc-
tory are: CPD.C, CPD.H, INCLUDES.H, CPD.CPL, TEST.C, and TEST.LNK.

- \$SOFTWARE\BLOCKS\SAMPLE\TEST
  This directory contains the pre-compiled DOS executable TEST.EXE. You can run this executable
  by opening a DOS window under either Windows 95, Windows 98, or Windows NT.
  This directory also contains a "batch" file (MAKETEST.INI) that will rebuild the object files
  using the Borland 'MAKE' utility and the 'makefile' TEST.MAK. Note that the makefile assumes
  that the Borland C/C++ compiler is located in the E:\BCDS\BIN directory but you can easily
  change that by editing TEST.MAK (see BORLAND and BORLAND.EXE in TEST.MAK).

- \$SOFTWARE\BLOCKS\SAMPLE\OBJ
  This directory contains the compiled object files for the building blocks that are used in TEST.EXE.
  You will find the following files in this directory:

  A10.OBJ
  CPS.OBJ
  CLK.OBJ
  COMM.CGS.OBJ
  COMM.CPL.OBJ
  DIO.OBJ
  KEY.OBJ
  LCD.OBJ
  OS_CPU.A.OBJ
  OS_CPU.C.OBJ
  PC.OBJ
  TEST.OBJ

- TEST.EXE
  TEST.MAK

  TEST11.OBJ contains the pre-compiled object code for μCOS-II. You can obtain the source code
  0-89791-543-6.
  OS_CPU.A.OBJ, OS_CPU.C.OBJ are the processor specific code for μCOS-II for an Intel (or
  AMD) 80x86. The code also supports hardware floating-point.

- \$SOFTWARE\BLOCKS\TIMER\SOURCE
  This directory contains the source code for the timer manager module (Chapter 7). The source files
  are TEST.C and TEST.H.

- \$SOFTWARE\HPLISTC
  This directory contains HPLISTC (Appendix D). The source file HPLISTC.C is found in \\
  \$SOFTWARE\HPLISTC\SOURCE. The DOS executable file HPLISTC.EXE is found in the \\
  \$SOFTWARE\HPLISTC\DOS directory.
E.03 Finding Errors

I have done everything I could to test the code provided in this book. If you find errors, I would like to know about them so that I can correct them or visit my web site at www.ucos-ii.com.

You can also contact me through R&D Books or by sending me a letter at:

Jean J. Labrosse
949 Crestview Circle
Weston, FL 33327
U.S.A.

E.04 Licensing

Embedded Systems Building Blocks (ESBB) source code and object code can be freely distributed (to students) by accredited colleges and universities without requiring a license, as long as there is no commercial application involved. In other words, no licensing is required if ESBB is used for educational use.

You must obtain an Object Code Distribution License to embed any ESBB code (i.e., module) in a commercial product. There will be a fee for each situation, and you will need to contact me for pricing. You must obtain an Source Code Distribution License to distribute ESBB's source code. Again, there is a fee for such a license, and you will need to contact me for pricing. You can contact me at Jean.Labrosse@UCOS-II.com or visit my web site at www.ucos-ii.com.

Write me at the address provided above, or call at:

(954) 217-2036
(954) 217-2037 (fax)
Index

Symbols
See also Appendix A

Numerics
80186 90, 96

A
ABSENT 255
abstraction
data 7
actuators 327
ADC 21, 328–331, 336–338, 344, 350, 365
address
  logical 256, 259
ADCfGain() 349
ADCfGainConv() 350
ADCfGainScaling() 372
ADCfGetA() 346, 354, 356
ADCfInit() 344, 355, 366
ADCfSetBypass() 356
ADCfSetBypassEn() 357
alarm check 191–192, 194, 202
alarm trips 195

American Standard Code for Information Inter-
change
See ASCII
amplifier 328, 340
analog 327
analog input channel 328
analog-to-digital
converter
See AOC
conversion 327–328
anode 134
AOCfGain() 358
AOCfGainConv() 359
AOCfGainScaling() 360
AOCfGetA() 347, 362
AOCfSetBypass() 363
AOCfSetBypassEn() 364
aperture time 330
API xiii, 402
Application Programming Interfaces
See API
ASCII 79, 402, 412
assembly language 96
asynchronous 62, 88, 278
asynchronous communications
See Chapter 11
asynchronous blanking 362, 266–267
auto-repeat 101, 104, 108
delay 104
B
backlighting 161
bargraph 173
baud rate 401, 404, 425
Baud Rate Generator 402
BCD (Binary Coded Decimal) 402
bilateral rendezvous 83
Binary Coded Decimal
See BCD
BIOS 497
BIOS Enable Select Switch 262, 278
blinking 261, 266, 278, 286
asynchronous 20, 262, 266–267
synchronous 20, 266, 285
breadboard 2
buffer 412
circular 412
ring 412–413, 417–418, 424, 434–436, 443
Bypass Switch 259, 262
C
calibrated
components 339
cathode 134–135
CPU.c 4, 6
cpu.h 4, 6
chainring the vectors 499
channel 328
analog input 344–346, 349
discrete input 258
discrete output 261
logical 258, 261
character 496
character LCD modules 163, 165
circular buffer 412
CLK.c 192
CLK.h 192, 206
CLK_DATE_LH 206
CLK_KLT_TICS 206
CLK_PASS_TICS 206
CLK_TS_EQ 206
CLK_USCL_DL 194, 206
CLK_FormatDate() 196, 198
CLK_FormatTime() 199
CLKGetTS() 200
CLKInit() 201
CLKMakeTS() 202
CLKSetDate() 203
CLKSetDateTime() 204
CLKSetTime() 205
CLKSetTimeObj() 194, 206
CLKTime() 193–194
CLKUpdateMax() 193
CLKUpdateOnOK() 193
CLKUpdateOnTime() 193
clock tick 69, 86, 94–96, 192, 194, 231–233, 244, 445, 450
clock/calendar 191–195, 206
clocks 191–195, 206
Cold Junction 370
CONTENT 423
and CONTENTD 453
and CONTENTD 452
ComIISI() 424, 432
ComIISI() 424, 432
and CONTENT 434
and CONTENT 452
ComConfig() 425
ComGetChar() 435, 437, 443, 445, 448
ComGetDiChar() 424, 436, 443
ComInit() 438, 447
ComWriteChar() 435, 439, 448
ComWriteNull() 440, 449
ComWriteHandler() 424, 434–436
ComWriteChar() 435, 441, 450
ComWriteChar() 424, 443
ComWriteIntVector() 433
and CONTENT 442
and CONTENT 453
ComWriteFlash() 427
ComWriteIntData() 428
ComWriteIntData() 429
ComWriteIntData() 430
ComWriteIntData() 471
communication 65, 80, 85
asynchronous
See Chapter 11
compensation
thermocouple 370
conditioning
input 328
configuration 2, 4, 6, 20
concurrent synchronization 84–85
context switch 65, 71, 77–78, 90
control register 164, 402
conversion
analog-to-digital 327–328
digital-to-analog 327, 340
speed 330, 340
time 330, 340
cooperative multitasking 66
countdown 240–243
time 230, 236
counting semaphore 77, 80-81
critical section of code 63
current-to-pressure transducer 340

D
DAC 21, 340–341, 344, 348, 359, 366
data
abstraction 2
communication protocols 400, 402
register 164
DCE 403–404, 406
deadlock 82
deadly embrace 82
debounce period 104, 107
delay 66, 71, 94–97
auto-repeat 104
delta list 232
deterministic 68
DT_MODE_3I 268, 286
cT_MODE_DIRECT 271
cT_MODE_EDGE_GPIO 271
cT_MODE_HIGH_GPIO 271
cT_MODE_LOW_G 271
cT_MODE_TOGGLE_HIGH_G 271
cT_MODE_TOGGLE_LOW_G 271
DTRACEEQDetectPinct 1269
DTRACEQMode 271
DUISet 257–258, 271, 274, 276–277

Digital to Analog Converter
See DAC

digital-to-analog conversion 32°, 340
Dijkstra, Edsger 77
DIO_TASK_DELAY_TICKS 265, 271
DIOUnit (i) 263, 275, 287
DIOUnit(i) 286–287
DID Check 263
DSiD(i) 263–264, 286
disable 75–76, 82, 85, 88, 93
scheduling 77
DISABLED 255
disabling interrupts 75, 82, 85, 88
discrete 255
input channel 258, 264–265, 286
inputs 255–259
output channel 261–262, 265–266, 267
outputs 256, 258–261, 263
DISetTimer ( ) 276–277
DISetTimer (i) 276–277
disjunctive synchronization 84
DSF_PLY_CNS 178
DSF_SEL_CMD_REG 178
DSF_SEL_DATA_REG 178
dispatcher 66
DisplayClear 168
DispChnlClear 169
DispChnlSet ( ) 140–141, 170
DispCurrent ( ) 178
DispDefaultClear ( ) 171
DispOldStatus 137
DispPurBck 173
DispPurBack 174–175
DispPurSet 140, 142, 146, 176
DispPromptPort 146, 178
displays 133
alphanumeric 162
character 162
custom 162
DispMsgDelete ( ) 146
DispMsgEnter ( ) 138, 146
DispMsgExit ( ) 146
DispMsgDelete 146
DispMsgEnter 137
DispMsgSet 178
DispMsgSetClr 140, 143
DispMsgSetSet 140, 144
DispMsgSet 139–140, 145, 177
E

E.U.

See Engineering Units

EBDIC (Extended Binary Coded Decimal Interchange Code) 402

EIA (Electronic Industries Association) 403

drivers/receivers 405

EL 161

electroluminescent light

See EL

Electronic Industries Association

See EIA

enable scheduling 71

ENABLED 255

encompass 79, 82

End Of Convexion signal

See EOC

Engineering Units (E.U.) 327, 337, 346

EOC 331–333

event flags 84

events 62, 66, 83–84, 88, 94, 97

exclusive access 63, 66, 68, 75, 79, 82, 85


execution time 62, 74, 90, 94–96

exponent 317–321

Extended Binary Coded Decimal Interchange Code

See EBCDIC

F

FALSE 255

feature 71, 82, 93, 97

FIFO 78, 86–87

filter 328, 340

low pass 328

fixed-point math

See Chapter 9

fixed-point numbers 315–317, 319–321

flag 82

flickering 135

floating-point 82, 96

arithmetic 344

hardware 1

math 315

numbers 177

flow control 412

PSV 341

diode 399, 402, 434

diodes 399

functions 2, 167

G

Gender Changer 406

ghosting 138

global variable 4

H

half-duplex 399, 408

heartbeat 94

Hitachi HD44780 LCD module controller 161, 163, 165, 168

HD44780

See Appendix D
I
I2 to P 340
I/O device polling 411
I/Os 255
IC 336
tID E viii
iBER-311
EXCLUDED: i 3.4, 6, 12
initialization 80, 88
input
anal og 25
conditioning 328
discrete 9, 19, 256-259
installation ESBR 1
instruction register 164
Integrated Circuit 336
Integrated Development Environment See IDE
interface
functions 2, 167
interrupt 63, 66, 69, 75-77, 82, 88, 90-94, 96-97
interrupt vector 66, 67, 76, 78, 82, 88-93, 78
mastering 89, 94, 97
response 89-90, 98
service routine 62, 87-88
Interrupt Vector Address 422
Interrupt-driven 400, 411, 420
interface communication 85
Invert Select Switch 262
ISR 62-63, 66-69, 76-78, 82-83, 85-93, 94, 422
IVT 432-433

J
jitter 94, 96
J-Type thermocouple 370

K
kernel 63, 65-73, 75-78, 82-84, 86-94, 96-98
key prefix 104-105
Shift 101, 104-106, 111
KEY_IOT 108
KEY_SCAN_TASK_KEY 107-108, 114, 116
keyboard matrix 104-105, 109-110, 114-115
module 115
scanning 101, 103-104, 106, 114
switch 302
KeyUpIn() 108
KeyUpDecode() 108
KeyUpList() 109-110, 116
KeyUpCol() 315
KeyUpGet() 108-109, 111, 116
KeyUpKeyPressDowntime() 109, 112, 116
KeyUpHit() 109, 113, 116
KeyUpInit() 107, 109, 114-115
KeyUpInitPort() 115
KeyUpScanTask() 106-108, 114-115
KeyUpSelRow() 115
keystroke 103-104

L
landscape 595
LCD 161-167, 171, 173, 176, 178
(line defined) 161
straight line 618
LCD C 165
LCD H 165, 178
LED xx, 133-136, 140, 143-144, 146-147
(defined) 133
displays, seven-segment 134-135
multiplexed 133
multiplexing 136
turning on 134
LED C 139
LED H 136, 146
LED A, A 136-137, 146
Light Emitting Diode See LED
linear bar graph 173
linked list 82
Liquid Crystal Display
See LCD
list
delta 232
literals 15
locked 77
logical
address 256, 259
channel 258, 261
low pass filter 328

M
m x n matrix keyboard 101
macro 75
mailbox 66, 85–87, 91, 97
mantissa 317–321, 323
mark 401
mask 257, 260
maskable 92
MASTER 408–409
MASTER/SLAVE 408
matrix keyboard 105–107, 109–110, 114–115
Maxim 7219 136
message
exchange 86
mailbox 86
queue 83, 87–88
MicroCASE-II, The Real-Time Kernel 7
microprocessor 82, 88, 91, 96, 98
Mode Select Switch 259, 262
module 162
countdown timer 229
keyboard 115
timer manager 230
momentary contact switch 101–102
multi-drop 408
multiplexer 328, 330, 332
multiplexing 137, 146–147
(defined) 135
LED 135
multitasking 63, 65–66, 69, 71, 77, 82, 96–97,
176–177
mutual exclusion 63, 66, 68, 73, 77–78, 82

N
n-key rollover 103
NM 91–94
node 408
L.D. 408
non-deterministic 62, 67
non-maskable interrupt 93–94
non-preemptive 66–67, 88–90, 92, 98
non-reentrant 66, 68–69
Neil Modem adapter 406

O
OFF 255–256
offline 330
ON 255–256, 263
OS_CPU.h 557
OS_ENTER_CRITICAL() 6, 75, 556
OS_EXIT_CRITICAL() 6, 75, 556
OSInit() 537
OSInitEnter() 90
OSInitExit() 90
OSOpenCreate() 538
OSGetPending() 78, 539
OSGetPos() 78, 541
OSStart() 543
OSStartInit() 544
OSTaskCreate() 545
OSTaskCreateEx() 548
OSTime64() 592
OSTime64y64() 553
OSVersion() 559
outputs 256
analog 21
discrete 9, 19, 256, 259–261, 263
overhead 65, 82, 89, 91, 94, 97

P
parameters
physical 528
party-line 408
pass count 350, 359
PC services
See Chapter 12
PC-C 495, 520
PC.I:495
PC_DispChart(): 502
PC_DispClrCol(): 503
PC_DispClrRow(): 504
PC_DispClrScr(): 505
PC_DispGrtzt(): 506
PC_DISPReturn(): 508
PC_DISPSaveReturn(): 509
PC_ELapsedInit(): 518
PC_ELapsedStart(): 16, 18
PC_ELapsedStop(): 17, 18, 513
PC_DateTime(): 514
PC_GetDay(): 515
PC_GetTickRate(): 516
PC_VectGet(): 537
PC_VectSet(): 518
PEND 77, 87-88
period
debounce 134
periodic 74
physical parameter 328
point-to-point interface 407
polling 420
the I/O device 411
portair 595
POST 77, 87-88
PPI 115, 163, 287
preempted 71
preemptive 66-69, 71-72, 74, 78, 88-90, 93, 97-98
prefix key 104-105
PRESENT 255
priority 63, 68-74, 77-78, 83, 86-88, 90-91, 94-96
inheritance 71-73
inversion 71-72
processing time 82, 91-92, 94
processor 62, 76, 82, 88, 90, 94, 97
Programmable Peripheral Interface
See PPI
PSW 422
push button 102
PVC8 572

\( Q \)

quantization size 329
quantizing 328
quantum 329
queues 66, 97

\( R \)

radix point 316-317
READY 63
real-time 61, 63, 65-67, 71, 73-74, 76, 88, 96-98
real-time kernel 414
reentrant function 68-69
registers 63, 65, 68, 88-89, 97, 410, 422, 424
control 164, 402
data 164
instruction 164
status 402
rendezvous 83-83
Resistance Temperature Device
See RTD
resolution 329, 337, 340
resource 63, 71-72, 74-77, 79-80, 82, 87, 57
response 62, 66-68, 76, 79-80, 89-93
responsiveness 67
RMS 74
rollover 103
n-key 103
round-robin scheduling 70
RS-232C 400, 403-404, 407, 412, 420
RS-485 399-400, 407-411
RTD (Resistance Temperature Device) 346, 369
RTOS 97-98
RUNNING 63

\( S \)

sample-and-hold 328
circuit 330
samples 330
scale factor 317
scaled 316-317
scan code 103-106, 108, 111, 116
scanning 350, 359
keyboard 101, 103-104, 106, 114
scheduler, 66, 77
  disabling 77
  enabling 77
Seebeck voltage, 370
services, 66, 84, 86–88, 90, 92–94, 97–98
settling time, 340
seven-segment, 140, 145
  LED displays, 134–135
lookup table, 139
shared resource, 63, 71–72, 75, 77
shift key, 101, 104–106, 111
signal, 77, 82–84, 91–92, 94
simplex, 399
SLAVE, 408–409
space, 401
speed
conversion, 330
stack, 63, 65, 68–69, 89, 97
start delay
auto-restart, 104
start signal, 401
state, 63
static priorities, 70
status register, 402, 411
stop signal, 401
structures, 65, 75–77, 97
suspend, 87–88
switch, 65, 69, 71, 77–78, 90, 94, 284
blink enable, 262, 278
bypass, 229
invert, 262
keyboard, 102
mode select, 259, 262
statement, 196, 198–199
synchronization, 82, 84–85
synchronous, 74, 77, 82–84
synchronous blanking, 262, 266, 285

T

TAS, 76–77
delayed, 94
multiple, 64–65, 69, 79, 84
priority, 70
response, 98
states, 65
switch, 65
  Test.C, 6, 8–9
  Test.CI, 6, 9
  Test.CI2G, 6
TEST, 105–7
TeensALOYTask(), 21
TestAndSet, 75–76
testC1Task(), 16
testC2DTask(), 19
testImplTask(), 15
TestInitModules(), 11–12
TestInitTask(), 22
TestInitTask2(), 11–12, 15
TestInitTask3(), 18
TestInitTask4(), 19
TestInitTask5(), 18
TestInitTask6(), 22
Texas Instruments, 408
thermocouple compensation, 370
THR (Transmitter Holding Register), 410–411
throttle, 327
tick, 94–96
time
  aperture, 330
  conversion, 330
timeout, 78–79, 82, 86–87
timers, 229
timestamp, 18, 191–195, 199–200, 202, 206
  (defined), 191
  UNIX_TICKS_SVID, 244
  UNIX_TIME, 234, 244
  UNIX_TIME, 244
  UNIX_TIMESTAMP, 244
  UNIX_USE, 244
  TimerCtrlTask, 234
  TimerChk(), 236
  TimerFormat(), 237
  TimerInit(), 238
  TimerReset(), 239
X
XON-XOFF 412
XON_XOFF 4

U
UART 456, 441
UART (Universal Asynchronous Receiver Trans-
mitter) xvi, 402, 409, 407, 410-411, 417-418,
422-424, 434-435, 443, 452
uCOS-III H 559
unilateral rendezvous 82-83
Universal Asynchronous Receiver Transmitter
See UART
user interface
code 117-118

V
V-(J Converter 341
variable
global 4
VC 572
'generic Control
See VC
voice coil 327
voltage
Seebeck 370

W
WAIT 78
WAITING 63
waiting 73-78, 83, 86-87, 94
www.ucos-ii.com.au